conference

proceedings

HotOS IX

The 9th Workshop on Hot Topics in Operating Systems

Lihue, HI, USA
May 18–21, 2003

Sponsored by
The USENIX Association
in cooperation with
The IEEE Technical Committee on Operating Systems (TCOS)

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX

Proceedings of HotOS IX

Lihue, HI, May 18-21, 2003

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

ISBN 1-931971-17-X

9th Workshop on Hot Topics in Operating Systems

HotOS IX

May 18–21, 2003
Lihue (Kauai), Hawaii

Sponsored by

USENIX, The Advanced Computing Systems Association,

in cooperation with the

IEEE Technical Committee on Operating Systems (TCOS)

Conference Organizers

Program Chair

Michael B. Jones, *Microsoft Research, Redmond*

Program Committee

Miguel Castro, *Microsoft Research, Cambridge*
Jeff Chase, *Duke University*
Armando Fox, *Stanford University*
Steve Gribble, *University of Washington*
Jeff Mogul, *HP Labs*
M. Satyanarayanan, *Carnegie Mellon University and Intel Research Pittsburgh*
Margo Seltzer, *Harvard University*
Geoff Voelker, *UC San Diego*
Dan S. Wallach, *Rice University*

External Reviewers

Tuomas Aura
Anand Balachandran
Ranjita Bhagwan
George Candea
Manuel Costa
Angela Dalton
Alexandra (Sasha) Fedorova
Robert Fischer
Tal Garfinkel
TJ Giuli
David Holland
Rebecca Isaacs
Terence Kelly
Emre Kiciman
Christos Kozyrakis
Kevin Lai
Monica Lam
Jonathan Ledlie
David Lowell
Kostas Magoutis
Priya Mahadevan
Robert Mayo
Dushyanth Narayanan
Chaki Ng
Tsuen-Wan "Johnny" Ngan
David Parkes
Trevor Pering
Patrick Reynolds
Rodrigo Rodrigues
Mema Roussopoulos
Antony Rowstron
Marc Shapiro
Jeff Shneidman
Alex C. Snoeren
Lex Stein
David Stewart
Kiran Tati
Mike Tucker
Amin Vahdat
David Wagner
Matt Welsh
John Wilkes
Beverly Yang

9th Workshop on Hot Topics in Operating Systems
May 18–21, 2003
Lihue (Kauai), Hawaii, USA

When Things Go Wrong

Performance Optimization

Storage 1

Trusting Hardware

Pervasive Computing

Storage 2

Index of Authors

Message from the Program Chair

I'd like to take this opportunity to thank all the participants in this year's HotOS workshop for helping make it a fun, informative, and highly interactive gathering of leading researchers in the systems field. We had great discussions, learned about exciting new work in its early stages, and often challenged one another's assumptions about what key challenges and opportunities are facing the field. A good time was had by all!

First, I'd like to thank the authors of all the submitted position papers. We accepted 32 out of the 144 submissions and received many more great position papers than we were able to accommodate in a 2 1/2 day program. Your creative research shows the vitality of the field, and I thank you again for submitting it to HotOS.

Next, I'd like to thank the program committee for all of their hard work in selecting our strong program from among such an ample selection of great submissions. Each program committee member reviewed 60 or more submissions with the dual goals of selecting the best ones for inclusion in the workshop and providing detailed, constructive feedback to all authors. I believe they did an exemplary job, and I'm grateful for their dedication and diligence. Thanks are also due to the external reviewers for their contributions to the paper selection process.

Many thanks are due to the many consummate professionals of the USENIX staff I worked with to make this year's HotOS come to life. Your attention to detail, planning, common sense, and good nature made running this workshop a breeze and a lot of fun! Particular thanks are due to Judy (and Paul!) DesHarnais, who handled all the details of the local arrangements, making sure that our setting of Lihue, Hawaii, on the island of Kauai was every bit the island paradise that it is—right down to the post-workshop mountain hike that Paul led for many of our outdoor-loving attendees.

My sincere thanks to HP Labs and Microsoft Research for funding student scholarships, allowing many students to attend at a substantially discounted price.

Andrew Hume opened us with a thought-provoking talk detailing some real-world experiences with operating system (un)reliability arising from his work with high-volume practical distributed data processing applications. His points kept being mentioned throughout the workshop!

Our four scribes did an admirable job recording the fast-paced give-and-take of the presentations and questions from the floor so that all can benefit from the discussions held there.

And finally, of course, thanks are due to all the presenters at the workshop for sharing your innovative "hot!" ideas with us, providing fodder for all the interesting discussions that ensued. Thanks for once again making HotOS *the* place where people learn about exciting, innovative new systems work!

Mahalo and Aloha!

Michael B. Jones
Redmond, Washington

Notes from the HotOS IX Workshop
May 18–21, 2003
Lihue (Kauai), Hawaii, USA

1. Invited Talk

Scribe: David Oppenheimer

1.1 Operating System Reliability: Does Anyone Care Any More?

Andrew Hume, *AT&T Labs–Research*

Andrew Hume of AT&T Research gave the HotOS keynote talk, entitled "Operating System Reliability: Does Anyone Care Any More?" He explained that his perspective comes from having designed, implemented, and delivered large-data applications for more than ten years. The problems he discussed in the talk were that operating systems have gone "from a help to a hindrance," that even users' lowered expectations for operating systems have not been met, and that, as a result, applications have to be designed around OS quirks. Hume pointed out that this situation hasn't always been the case, citing WORM-based backup systems in research versions of UNIX and a cluster-based billing system that AT&T built using Plan 9 as examples of systems that were highly reliable, even under load.

The first problematic system Hume described was Gecko, a large-scale (250GB/day) billing system implemented in 1996 on Solaris 2.6. AT&T required 1 GB/s of filesystem throughput and predictable use of memory. Among the problems encountered were that Solaris crashed every few days for the first six months that the system was in production, that Solaris limited file throughput to about 600 MB/sec, that reading large files sequentially crashed the VM, and that a "VM roller coaster" developed when a large chunk of memory was allocated (resulting in all of physical memory being paged out and then almost all of the same data being paged back in, over and over again in a cycle, rather than the system just paging out the amount of new memory needed).

The second problematic system Hume described was a replacement for Gecko that required 6 times the capacity of the original Gecko. This system was implemented on a cluster running Linux. The architecture was a "Swiss canton" model of loosely affiliated independent nodes with a single locus of control, data replication among nodes, and a single error path so that software could only halt by crashing (there was no explicit shutdown operation). Hume described eight problems the Gecko implementors experienced with Linux (versions 4.18 through 4.20), including Linux's forcing all I/O through a filesystem buffer cache with highly unpredictable performance scaling (30 MB/s to write to one filesystem at a time, 2 MB/s to write to two at a time), general I/O flakiness (1-5% of the time corrupting data read into gzip), TCP/IP networking that was slow and behaved poorly under overload, lack of a good filesystem, nodes that didn't survive two reboots, and slow operation of some I/O utilities such as df. In general, Hume said, he has concluded that "Linux is good if you want to run Apache or compile the kernel. Every other application is suspect."

Hume proposed the following definition of OS reliability: "[The OS] does what you ask, or it fails within a modest bounded time." He noted that FreeBSD has comparable functionality to Linux, better performance, and higher reliability, and he speculated that this might stem from BSD's (and other "clean, lean, effective systems") having been built using "a small set of principles extensively used, and a sense of taste of what is good practice, clearly articulated by a small team of mature, experienced people." Hume took Linux to task for not demonstrating these characteristics, in particular for being too bloated in terms of features, and for having been developed by too large a team. Further, he singled out the Carrier Grade Linux effort for special condemnation for "addressing zero of the [type of] problems" he has had.

1.2 General Discussion

During Q&A, Margo Seltzer asked if perhaps part of the problem is that the dependable systems community has traditionally been isolated from the operating systems community. Hume agreed, and he suggested that the dependability community may understand some things that the OS community does not yet, such as the end-to-endness of the reliability issue. Armando Fox asked if perhaps the problem is not the sheer number of developers who contribute to projects that have turned out imperfect but what many users perceive as "good enough" systems (e.g., Linux and HTTP), but, rather, the need for invariants to which all the developers would program, to help make such systems more reliable. Fox suggested Hume's rule of "It does what you ask or it fails within a modest bounded time" as one possible invariant. Hume agreed, saying that he is working hard on educating the Linux community about

how to make the OS more reliable, particularly in large-scale HA configurations like that used by Gecko. Jeff Mogul suggested that perhaps what's missing is repeatable, easy-to-run OS dependability benchmarks based on the kinds of failure anecdotes Hume was describing. Jay Lepreau suggested that instead of taking a large general-purpose OS like Linux and stripping it down to a reliable core, an alternative approach would be to start with a reliable application-specific OS and to add just the extra functionality that might be necessary to run Gecko. Hume disagreed, suggesting that he had more hope for making existing general-purpose OSes more reliable. Tal Garfinkel pointed out that Hume had talked primarily about performance scalability rather than absolute reliability, and Hume agreed that for his application absolute single-node reliability is less important than the ability to get predictable performance scalability when more nodes are added and predictable performance degradation when nodes fail. Ethan Miller asked if product liability lawsuits might help the OS reliability problem, but Hume disagreed, stating that he is "not keen on litigation to direct OS research." Finally, Val Henson asked if OS reliability, while bad, is perhaps "good enough" for most people. Hume agreed, but suggested that we should raise our expectations.

2. The Emperor's Clothes

Chair: Steve Gribble
Scribe: Matt Welsh

2.1 High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two

Charles Blake and Rodrigo Rodrigues, *MIT Laboratory for Computer Science*

Charles Blake opened the first technical session with a talk on the overhead of "maintenance bandwidth"—network bandwidth consumed to maintain a given level of replication or redundancy—in a peer-to-peer storage system. The basic argument is that maintenance bandwidth across the WAN, not the aggregate local disk space, is the fundamental limit to scalability in these systems. Given the dynamics of nodes joining and leaving the system, Charles presented a conservative estimate of the maintenance bandwidth that scales with the WAN bandwidth and average lifetime of nodes in the system. Under a typical scenario (100 million cable modems with a certain bandwidth available for replication, one week average lifetime, and 100 GB storage per node), only 500 MB of space per node is usable, only 0.5% of the total.

To try to address these problems, Charles looked at alternatives such as admission control (only admitting "reliable" nodes) or incentivizing nodes to have long lifetimes. It turns out that a small core of reliable nodes (such as a few hundred universities with a single reliable machine dedicated to hosting data) yields as much maintenance bandwidth reduction as millions of home users with flaky connections. The talk concluded with a number of open issues in organizing WAN-based storage systems, such as whether it is appropriate to assume millions of flaky users and whether the requirement of aggregate data availability should be reconsidered.

2.1.1. Discussion

Armando Fox raised the point that, rather than high availability, millions of home users really bring lack of accountability. The question is whether there is any benefit other than pirating movies, and Charles answered that perhaps circumventing legal requirements should be a top design goal of these systems.

Eric Brewer argued that once a system becomes popular, it will be centralized, but that P2P systems are really useful for prototyping. Charles responded that, while there are benefits to collaboration, one may as well use a small number of available, well-connected/well-motivated friends to do a prototype.

Margo Seltzer wondered whether P2P could be useful to the Grid community and their interest in harvesting cycles rather than storage. Charles agreed. Andrew Hume pointed out that the Grid community is talking about needing gigabits of networking between sites.

Mike Swift wondered about not providing a disconnect button in the user interface, as a way of creating an incentive to users to offer long lifetimes.

Timothy Roscoe asked whether the tradeoff between storage space and bandwidth usage was inherent, or whether it was a transient condition resulting from current economic conditions. Charles replied that telcos may not have the motivation to increase bandwidth upstream from home users if most applications (e.g., Web surfing) require only a small amount.

2.2 One Hop Lookups for Peer-to-Peer Overlaps

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues, *MIT Laboratory for Computer Science*

Anjali Gupta presented a talk on the use of one-hop lookups in peer-to-peer systems, avoiding the high latency associated with the typical log(N) lookup paths required by most systems. The challenge is keeping up with membership change information on all hosts. For example, the UW Gnutella study in 2002 showed an average node session time of 2.9 hours, implying 20 membership changes per second in a system with 100,000 hosts. Anjali presented a hierarchical scheme,

in which the address space (forming a ring) is subdivided into slices, each with a slice leader that is the successor to the midpoint in the slice. Slices are further subdivided into units.

The basic approach is for nodes to exchange frequent keep-alive messages with their predecessor and successor nodes. A change to a node's successor is an event that is propagated by piggybacking a recent event log onto keep-alive messages. A node change event is relayed to the slice leader, which periodically (every 30 seconds) notifies other slices of the updates. Internally to a slice, slice leaders periodically (every 5 seconds) notify unit leaders of node change information. Given some reasonable assumptions on the size of the system, all nodes can be updated within 45 seconds of a node leaving or joining the system, which permits a 99% "hit rate" for an address lookup. In this scheme, it is important to choose good slice leaders that are well-provisioned. Anjali concluded with a summary of ongoing implementation and experimentation work, noting that systems larger than a million nodes will require two-hop lookups.

2.2.1 Discussion

Mike Jones noted that there would most likely be a burst of traffic to and from a new node as it comes up, and he asked how a node just joining could communicate when it would take the 45-second update propagation delay before other nodes would know how to reach the new node. Anjali responded that a node starts participating only after it completely downloads a routing table from a unit or slice leader.

Dan Wallach asked about correlated failures, where a node and its successor fail at the same time. Anjali responded that the odds of this occurring are low and that this only introduces a bit more delay for the update. Dan asked about malicious nodes, and Anjali responded that this is not addressed by this work.

Ethan Miller asked about how the system scales up to a million-node system; Anjali said that this system reaches its upper limit at that size and does not scale beyond it.

Timothy Roscoe asked whether this system assumes that every node can talk to every other node in the network directly, and whether this was realistic. The response was that they were indeed making the assumption that every node can talk to its neighbors and unit and slice leaders. There are cases when this assumption may be invalid, and this question will be addressed in future work.

Eric Brewer pointed out that there is a lot of work in industry on hierarchical caching, allowing a big win since updates need not be propagated unless an object is hot. Anjali said that they were looking at designs where information was propagated for all objects, and had not looked at schemes that would do bookkeeping for object popularity. Nevertheless, it was a good idea and worth looking at.

2.3 An Analysis of Compare-by-Hash

Val Henson, *Sun Microsystems*

Val Henson presented one of the most controversial papers of the conference, admonishing those systems that rely upon comparison of data by comparing cryptographic hashes of the data. Many systems (such as rsync, Venti, Pastiche, LBFS, and OpenCM) use this technique, but it is not yet widely accepted by OS researchers, due to little characterization of the technique and many unanswered questions. The risk of collision using (say) a 160-bit SHA-1 hash is just 2^-160, which is far lower than a hardware failure or probability of an undetected TCP error. So why the controversy?

First, these techniques assume that data is random, but real data is not random and has a fair amount of commonality (think about ELF headers and English text). Second, cryptographic hashes were designed for authentication and care about "meaningful" collisions, such as two contracts with the same text but different dollar amounts that happen to collide in the hash space. Third, hash algorithms are short-lived and obsolescence is inevitable—systems need an upgrade strategy. Finally, collisions are deterministic—two blocks that collide always collide—rather than a transient error such as a hardware fault. Hash collision is therefore a silent error in those systems that rely on compare-by-hash techniques. Val claims that we should be striving for correctness in systems software, not introducing "known bugs." It is OK to rely on compare-by-hash when the address space is not shared by untrusted parties, and when the user knows and expects the possibility of incorrect behavior, and he cited rsync as an example. Note that "double hashing" is not an acceptable solution, as this results in just another hash function, albeit one with a lower collision probability.

Some alternatives to compare by hash were discussed, such as content-based addressing that checks for collisions; using compression maintaining state only to send or store identical blocks once (as in LBFS); sending diffs instead of an entire block; or using universal IDs for common blocks.

2.3.1 Discussion

Peter Druschel argued that Val was taking a purist view of correctness, and that optimizations in general lead to complexity which reduces reliability in systems. Val responded that this has to do with tradeoffs, and that the tradeoffs taken in compare-by-hash were not the right ones.

Tal Garfinkel claimed that Val's arguments were based on "math FUD" and that paranoia is unwarranted—after all, all of cryptography is based on this technique. Val responded that we should not assume that hash functions designed for cryptography are appropriate for these applications. Tal further explained that similar data does *not* produce similar hashes, contrary to her assumption.

Andrew Hume pointed out that checksums work fine as long as there is a uniform distribution across the hash space. He also commented that while he is skeptical of Val's birthday paradox argument that with 2^80 documents there is 50% chance of a collision with a 160-bit hash, that if true, that is a very scary probability and that a lot of people would be much happier with a much smaller threshold like one in a million or one in a billion. Val answered that this is a difficult number to calculate (due to the infinitesimal quantities involved), and indeed, that teams of people and weeks of time could not produce the standard "M = sqrt(2 * p * N)" formula for the address space pressure needed to produce a collision with probability p, but that they believe it.

Dirk Grunwald asked at what point the hash collision probability becomes infinitesimal compared to everything else, such as TCP or ECC bit errors. Val again answered that the seriousness of the undetected collision made it important to consider the possibility.

2.4 Why Events are a Bad Idea (for High-Concurrency Servers)

Rob von Behren, Jeremy Condit, and Eric Brewer, *UC Berkeley*

Rob von Behren raised the argument of thread-based versus event-driven concurrency in high-concurrency servers, claiming that thread-based approaches are far better due to their ease of programming. To counter the arguments that threaded systems have inherently higher overhead than event-driven ones, Rob presented early results from a lightweight user-level thread system that performed as well as an event-driven system on a Web server benchmark. Furthermore, threads have better programming and debugging tools, leading to increased productivity. To address the problem of high overhead for per-thread stacks, Rob proposed the use of compiler support to automatically compress stacks, for example, by moving "dead" elements off the stack across a blocking operation. Using cooperative scheduling avoids the overhead of generic thread synchronization; however, there are some issues to address here such as fairness, the use of multiprocessors, and how to handle spontaneous blocking events such as page faults.

Rob pointed out that events have the advantage of permitting very complex control flow structures; however, very few programmers use these structures, and threads can capture the vast majority of scenarios. Another problem with thread schedulers is that they are "generic" and have little knowledge of application structure. To permit greater cache locality, Rob proposed "2D" batch scheduling, in which the compiler annotates the application code to indicate to the scheduler system where the various stages of the thread's execution are located.

Rob presented some measurements of a simple Web server benchmark based on his user-level threads package, capable of supporting over 100,000 threads, implemented in about 5000 lines of C code. The server outperforms a Java-based event-driven Web server, probably due to the large number of context switches in the event-driven system. Rob concluded that it may be possible to achieve higher performance using threads than events, in part because events require dynamic dispatch through function pointers, which makes it difficult to perform inlining and branch prediction.

2.4.1 Discussion

Timothy Roscoe shared a vignette from his upbringing: "When I was young, I was surrounded by big tall men with beards, who talked about something called continuations." He asked whether this is what Rob was proposing and whether we shouldn't be looking at what the functional languages community is doing in this area. Rob was brief—"Yes."

Margo Seltzer asked about the background of this work, to which Rob explained that the UC Berkeley Ninja project had first gone down the path of using threads to build a large-scale system, and then switched to events to address some of the scalability problems. Both approaches "kind of sucked," and when Rob started writing tools to fix this problem he realized that a lean, user-level threading approach was the right solution.

Jeff Chase asked about writing tools to make it easier to write in an explicit event-based model, rather than a thread-based model. His motivating example was distributed systems using callbacks, for which an event-driven style of programming seems inevitable. Rob responded that one does not need to use events to handle callbacks—with cheap enough threads, one can spawn a new thread to wait for the callback.

2.5 General Discussion

Charles Blake kicked it off by asking why Val's birthday paradox probability was so hard to compute. She responded that essentially it comes down to the infinitesimal numbers involved. Afterwards Charles

contended that while this was Val's response, that has nothing to do with the actual state of affairs, since the next order term in M=sqrt(2pN) is proportional to O((M/N)^2), which is very negligible on any of the address-space pressure scales in question.

Eric Brewer pointed out that systems should use CRC, not MD5—since CRC is no good for preventing malicious collisions, there should be no illusion that it is. One should also use a random salt with the checksum, which should help with the non-randomness of real data. Val responded that if one has to recompute the checksum across the actual data, then you are losing the benefits of this technique.

George Candea raised the point that, although a P2P client that prevents a user from disconnecting appears less desirable at first, it would lead to higher availability for the service as a whole. This makes the service more valuable, and hence provide greater incentive to use it (i.e., download the client).

Ethan Miller asked whether people are really comfortable with the concept of probabilistic storage. Val agreed that the notion of dynamic, unreliable storage systems makes her uncomfortable.

Ranjita Bhagwan pointed out that Charles's calculations don't push P2P out of the picture, asking whether there may be a cost benefit to a peer-to-peer approach versus a centralized approach. Charles said that fundamentally his argument was economic, concerning the bandwidth versus storage requirement for these systems. Andrew Hume said that the best nodes are professionally managed and that high-bandwidth connections and support are expensive, so the economics of the two approaches are more similar than they are different.

Mohan Rajagopalan said that compiler optimizations actually perform very well for event-based systems, and that implementation is what really matters. Event-based systems permit a decoupling between the caller and callee, so it is easier to write an event-based "adaptive" program than one in threads. Isn't this a fundamental benefit? Rob responded that events do make it easier to perform composition and interpositioning, but that this can also be done in the thread model. Eric Brewer mentioned that Click is very configurable and runs as a single large thread.

Peter Druschel was skeptical that we can do P2P storage based on home connected desktops, but that the alternative is not centralized systems. For example, one can reap the benefits of unused desktop systems within a large organization. Charles did not disagree with that.

This was followed by an exchange between Peter and Rodrigo Rodrigues about using so-called "scalable lookup" (Pastry/Chord/CAN/...) vs. some other organization for P2P file storage. Basically Rodrigo pointed out that in a scenario where the individual nodes are very available/reliable and the network isn't giant, there is no need for scalable lookup and other considerations should take higher priority. Peter responded that having a large number of nodes and security implied the need for small lookup-state optimizations.

3. Popping & Pushing the Stack

Chair: Geoff Voelker
Scribe: Ranjita Bhagwan

3.1 TCP Offload Is a Dumb Idea Whose Time Has Come

Jeffrey C. Mogul, *Hewlett Packard Laboratories*

TCP offload in the traditional sense violates performance requirements, has practical deployment issues, and targets the wrong applications. TOEs impose complex interfaces and cause suboptimal buffer managemnt. Moreover, lots of small connections overhelm savings because of connection management. Event management is a problem. Lots of virtual resources need to be managed. Also, one of the main motivations for TOE has been that TCP implementation in the OS is bad.

However, it is no longer a dumb idea, because now we are offloading higher level protocols onto hardware. The justification for offloading TCP is simply that you can't offload the higher level protocols without also offloading TCP. The sweet spot for TCP offload is when the application uses very high bandwidth, has relatively low end-to-end latency, long connection durations, and relatively few connections. For example, storage server access and graphics. Also, several economic trends are favourable towards TCP offload. One would like to replace special purpose hardware with cheap commodity parts, such as 1 gig or 10 gig ethernet. This helps because with these in place, operators have only one kind of fabric to provision, connect and manage. Still, many challenges remain. Data copy costs still dominate, and busses are too slow. Zero copy and single copy seem too hard to adopt in commercial OSes. However, with the advent of RDMA, vendors want to ship RNICs in volume, allowing one kind of chip for all applications. It would mean cheaper hardware. Also, there are several upper level protocols available, such as NFSv4, and DAFS. Still, many problems of TCP offload still apply. There are security concerns, and so far the benefits have been elusive. The new networking model may require changes to traditional OS APIs. Systems people need to give this due consideration.

3.1.1 Discussion

Margo Seltzer observed that by coupling RDMA and TCP in hardware, some of these issues may be resolved since now you would get a generic version of RDMA. Jeff said that offloading even higher level protocols is possible, but is at the same time difficult.

Rob Von Behren asked what the write interface should be like. Jeff responded that standardization of APIs is not a priority. Currently, the concentration is on standardizion of primitives. However, he does not know if that is sufficent.

Val Henson said that linux has a commerically viable implemenation of zero-copy, that uses TCP checksum implemenation in the NIC.

3.2 TCP Meets Mobile Code

Parveen Patel, *University of Utah*; David Wetherall, *University of Washington*; Jay Lepreau, *University of Utah*; Andrew Whitaker, *University of Washington*

The authors address the problem of deployment of transport protocols, by proposing an extensible transport layer, called XTCP. The main argument is that transport protocols, such as TCP, need a self-upgrade mechanism and untrusted mobile code can help build such a mechanism. Several modifications to TCP, as well as alternative transport protocols have been proposed. However, as with any new protocol, deployment is an issue. Currently, it takes many years before a new protocol or an extension can be used by applications: a new protocol or extension has to be approved by standards committees, implemented by OS vendors and finally enabled-by-default at both ends of communication.

In the proposed solution, untrusted peers can upgrade each other with new transport protocols using mobile code. A typical usage scenario is that of a web server. A web server can download a high-performance version of TCP, after which it tells every client to download the same version from it. Then the client and the server can speak the upgraded version of TCP. This solution avoids all the steps of the deployment process that need approval and support from third parties, such as standards committee and OS vendors.

There are several challenges to building such an extensible layer, especially of host and network safety. The presenter contrasted XTCP with "active networking" and argued that the domain of transport protocols is restricted enough that host and network safety challenges can be met without degrading performance.

Host safety is assured by providing memory protection and resource control. Memory protection is achieved by using Cyclone, a typesafe C-like language. The stylized memory usage pattern of TCP extensions—no shared state between extensions and predictable ownership of data buffers—makes it resource control possible using traditional runtime methods. XTCP uses the well-understood notion of TCP-friendliness as a measure of network safety. All extensions written using the XTCP framework are forced to conform to TCP-friendliness using the ECN nonce mechanism. In constrast, active networking had no such well-defined notion of network safety, and host safety in the face of arbitrary code was costly.

XTCP has been implemented in FreeBSD 4.7. Support for user-level transports is being developed currently.

3.2.1 Discussion

Constantine Sapuntzakis asked how many extensions could be implemented on the XTCP API. Parveen said that 25 could be. The other two that they considered were non TCP-friendly.

Peter Steenkiste asked who builds the headers. Parveen responded that XTCP builds the IP header while the extension builds the transport header. Peter then asked that if the definition of TCP friendliness changes, how would XTCP work? The speaker said that currently, changing the definiton of TCP friendliness would require a regular upgrade of the operating system.

Val Henson said that she was not convinced that XTCP could be secure. She thought extensions can be malicious or non-performing in more ways than stated, although she lacked specific examples. Parveen responded that non-performance of extensions is not a threat to XTCP. This is because extensions can only hurt or benefit the code provider. For example, if Yahoo accepts code from a client, then that extension will be used only in sessions with that client. So the extension provider has an incentive to provide good code.

Geoff Voelker asked what would happen in the case of worm spread. One can compromise a server such as yahoo.com and then use this mechanism to spread worms. Each worm will behave in a TCP-friendly manner but many worms can collaborate and attack a remote host. Parveen responded that such a worm could not be implemented using XTCP alone, because the IP header is built by the XTCP layer, and the extensions cannot affect it. Therefore, extensions cannot send data to an arbitrary host unless an application explicitly opens a connection to it and sends data. Furthermore, an extension received from yahoo.com will be used for communication with yahoo.com alone. That makes XTCP as harmless or harmful as the currently deployed trusted versions of TCP.

3.3 Exploiting the Synergy between Peer-to-Peer and Mobile Ad Hoc Networks

Y. Charlie Hu, Saumitra M. Das, and Himabindu Pucha, *Purdue University*

There appears to be a number of similarities in the problems that research in peer-to-peer and mobile ad-hoc networking address. One such area is that of routing. The speaker showed the similarity between the problems solved by Pastry and how it can be used in ad-hoc networking too. He described a new protocol, DPSR, which stores routing state in a manner similar to Pastry's. This reduces routing state per node from O(N) to O(log N). DPSR uses node ID assignment, node state, routing, node join procedures, and node failure or out of reach in much the same manner as Pastry, inherits all DSR optimizations on source routes, and contains a number of additional optimizations related to Pastry's routing structures and operations.

Simulations of DPSR for a 50 node system show that the routing overhead of DPSR scales better than that of DSR. In short, DPSR outperforms DSR when the number of connections per source is greater than 1. It ties DSR otherwise.

3.3.1 Discussion

Timothy Roscoe asked how many routes were going through the pastry routing for more than one hop. Charlie said that though he did not specifically measure this number, the average number of overlay hops for the 50 node simulation is around 1.5.

Srini Seshan asked why not do a low TTL flood optimization to DSR. Charlie said this does not help in general.

3.4 General Discussion

Bogdan Popescu asked Parveen (XTCP) if you could use a signing mechanism to detect unresponsive connections. Parveen said that the nice thing about XTCP is that it works well without it. Bogdan said that then you could have DoS attacks.

Rob Von Behren said that it would be very easy to do DoS on XTCP, such as malloc'ing large amounts of memory, using a lot of CPU time, etc. Parveen said that each malloc call is accounted for. Rob responded that there is the problem of DDoS. With a considerable number of nodes using a little too much memory, one could perform a DDoS attack. Parveen said that this is possible; the only way to avoid it is strict admission control.

Jay Lepreau brought up Geoff Voelker's question again, and said that the code is identified by a hash. One could develop a history-based code-rating scheme, so that if a specifically "bad" user tried to use an XTCP extension, the server would disallow it. But then that user could keep changing the code slightly.

Jeff Mogul said that you have to make sure that XTCP itself is not subvertible. Because if it is, then it is a very rich environment for spreading worms.

Peter Steenkiste said that in the early '90s, after 6 months of effort, he had decided that TCP offloading is no good. In general, enthusiasm for TCP offload seems lukewarm. He asked Jeff if it would take off. Jeff responded that he does believe that it will take off, mainly for commercial reasons. Having only one fabric to manage for datacenters seems good. Peter said that there appears to be a contradiction somewhere here. Earlier on, we wanted to move things to the software level, and now attempts are being made to move them to hardware. Jeff said that switches are clearly a larger investment than NICs, so commoditizing the NICs would be good.

Geoff Voelker asked Charlie about the benefits of his approach depending on the number of shared source routes. Did he have a sense of the minimum shared source routes needed for DPSR to work? Charlie answered that so far, the sharing was small, but even in this scenario, DPSR does not do worse than DSR, so it seems overall to be a gain over DSR.

4. Distributed Systems

Chair: Jeff Chase
Scribe: Amit Purohit

4.1 Scheduling and Simulation: How to Upgrade Distributed Systems

Sameer Ajmani and Barbara Liskov, *MIT Laboratory for Computer Science*; Liuba Shrira, *Brandeis University*

In the first talk of the session Sameer presented a solution for upgrading distributed software automatically with minimal service disruption. He presented a technique which uses a combination of centralized and distributed components. The infrastructure consists of three main components. Scheduling functions tell the node when to upgrade. Simulation objects enable communication among nodes running different versions. Transform functions change a node's persistent state from one version to a higher one.

Timothy Roscoe pointed out that this method is expensive. He claimed that instead of upgrading each node and preserving the state, it is possible to ignore the state. Sameer agreed that one can ignore the state and allow the system to recover it, but said that this reduces the rate at which upgrades can happen and may require additional state transfer over the network. A. Veitch

asked Sameer to compare his work with the existing work on on-line upgrades. Sameer said they are mainly targeting distributed systems.

4.2 Development Tools for Distributed Applications

Mukesh Agrawal and Srinivasan Seshan,
Carnegie Mellon University

In the second talk of the session Mukesh explained the motivation for his current research. He claimed that there are few distributed applications, not because they are not useful, but because they are very hard to implement. He cited routing table upgrading for distributed applications as one of the hard problems. He mentioned ns-2 simulator as a tool that helps to compare design choices. DHT is developing building blocks to help in implementation of distributed systems. Mukesh then pointed out some inherent flaws in the current approaches. The main concentration of the research is on the initial stages of the life-cycle of the applications, whereas his work mainly addresses the later life-cycle stages of the application.

4.2.1 Discussion

George Candea asked whether there is any good reason to build a distributed application instead of a centralized one. Mukesh answered that control over the application is the key. George argued it is not safe to give up control, to which Mukesh replied a single trusted entity is not necessary.

4.3 Virtual Appliances in the Collective: A Road to Hassle-free Computing

Constantine Sapuntzakis and Monica S. Lam,
Stanford University

In this talk Constantine envisioned a computing utility that runs not only Internet services but highly interactive applications that are commonly run on desktop computers. Patches arrive at a high rate. Multiple applications share a lot of elements, such as the OS and shared libraries. Hence, an upgrade to one application can break another. He argued that it is possible to borrow idea from appliances to improve the manageability and usability of computers. Then he described the architecture of their framework. Appliances are maintained by makers without user involvement. Cheaper hardware made virtualization an attractive option.

4.3.1 Discussion

Jay Lepreau was concerned about the deployment of this idea in wide use. He was skeptical about whether people would give up their autonomy. He said it may be advantageous to separate the application into sets but even that is not problem-free. Constantine replied that the appliance could be fairly small, making its deployment easier and reliable. David Oppenheimer asked how this would differ from application services servers. Constantine said it is more flexible than the centralized techniques. Andrew Hume joined the discussion at the end. He said it is very important that the interface be extremely simple. This is a real challenge, since appliances are dumb. Constantine is offering a simple user interface; hence it is not a complex mechanism, so it is hoped that it will be easy to use and fun for the user.

4.4 POST: A Secure, Resilient, Cooperative Messaging System

Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, and Dan S. Wallach,
Rice University*; Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra,** ***Université Paris VI

The author presented a p2p solution that interoperated seamlessly with a wide range of collaborative services by providing one serverless platform. It provides three basic services to applications: a secure single copy message storage; an event notification service which alerts users to certain events such as availability of a message; and single writer logs, which allow applications to maintain metadata. The author claims that their features are sufficient to support a variety of collaborative applications.

4.4.1 Discussion

Mike Jones argued that application-level multicast will achieve the same goals. The author said the main motivation is to prevent spam. Armando Fox asked, what is the motivation behind choosing email as the primary application. The author replied, email has great locality properties. If such a demanding application can be changed, then POST can be extended to more complex applications.

4.5 General Discussion

In the panel session, M. Swift asked Constantine about the cost of complexity. Also, device drivers talk to hardware, so they couldn't be virtualized. Moreover, if they are buggy, then they can crash. Constantine said in future device drivers would be written in user land, then the problem could be solved. But if the application crashes, not much can be done.

Eric Brewer stated the view that the hard part is sharing information: having separate virtual appliances for everything only works if they don't share any information, which means that the user must replicated all "shared" information by hand (as we do now with real appliances, e.g., setting the clock). The path of safe sharing leads you to shared segments as in Multics,

including layers (rings) and call gates for protected calls into shared resources. The author replied that Multics has some problems, which they are planning to address as well.

5. Outrageous Opinions Session

Chair: Jeff Mogul
Scribes: David Oppenheimer and Matt Welsh

5.1 Scribe Notes by David Oppenheimer

The Outrageous Opinions session featured 13 speakers.

Val Henson kicked off the session by suggesting that end-to-end checksums replace component-to-component checksums. Andrew Hume disagreed with this philosophy, pointing out that component-to-component checks are vital for pointing the finger at the component that has caused the problem.

Matt Welsh presented "a brief history of computing systems research," in which he urged computer scientists to think about how their research can help to address social problems. He pointed out that computer scientists have always worked on improving life for computer scientists, focusing on improving their own day-to-day tasks. He suggested that computer scientists should think more about social problems rather than "How can I download porn and pirate music more efficiently?" In particular, he cited education and health care, particularly outside the United States and Europe, as social problems that computer scientists could help tackle, for example by empowering local government and remote communities. Specific technologies he cited were peer-to-peer wireless networks for emergency and disaster response systems; censorship-proof electronic publishing of dissenting political opinions; sensor networks for environmental monitoring, precision agriculture; inexpensive medical diagnostics; highly reliable power environments; and maintenance-free PCs.

Mendel Rosenblum talked about the similarities between microkernels and virtual machine monitors (VMMs) for running multiple untrusted environments on top of an operating system. Both provide isolation, resource management, and a small operating environment with simple interfaces, leading to high assurance. The key difference is what runs on top of each—for a VMM you don't need to port applications to run on it, whereas for a microkernel you do. He pointed out that despite this advantage for VMMs, academics and industry researchers are often interested in microkernels, because by not leveraging existing software, microkernels provide academics an opportunity to write lots of papers and industry an opportunity to define APIs, leading to an industry control point.

Mike Chen presented "two easy techniques to improve MTTR": redirect users' attention to what's still available when something becomes unavailable (e.g., "Please read this new privacy policy"), and blame it on the user (e.g., "Did you forget your password? Please try again."). He pointed out that by tricking the user, perceived availability can easily be increased.

Timothy Roscoe presented a short talk entitled "Scalability Considered Harmful." The core of his argument was a rejection of the traditional dogma that maximum scalability is always a good thing. Instead, he argued that systems should scale as far as required, but no further. He argued that extra scalability is bad because it facilitates abuse and concentrates power. Moreover, he claimed that non-scalable systems are always more efficient than scalable systems. As an example, he pointed out that email scales globally, but it doesn't need to—the real social communication graph is cliquey, but because email scales beyond the group of people with whom a user really wants to communicate, we have the problem of spam. A second example he cited was one he credited to Chris Overton: an unscalable mechanism that some societies use to solve trust problems is reputation and referral, which works, while a scalable mechanism that some societies use to solve trust problems is lawyers, which doesn't work. A third example Roscoe gave was that of email whitelists versus blacklists—whitelists don't scale but are highly effective at preventing spam, while blacklists do scale but are much less effective. A final example he cited was Google, a highly scalable, indispensable system. He argued that the downsides of Google's success are that they have a lot of power, it's not possible for someone else to deploy a competing search engine anymore, and paper-based publishing and libraries have declined. The non-scalable alternative he argued for instead of Google was traditional libraries. Roscoe conclude by summarizing that scalable systems benefit the system owners to the detriment of the system users. He argued that by designing scalable systems, we as researchers collude with the concentration of power in the hands of the few. He therefore suggested that researchers stop designing scalable systems.

Dan Wallach argued that the logical extension of the virtual machine is the process.

Eric Brewer issued a call for IT research for developing regions. He made 5 claims: (1) there is a need for a direct attack by developing new devices rather than giving developing regions hand-me-down machines, which are a bad fit on cost, power, user knowledge, administration, and user literacy; (2) there is a need for infrastructure to support thousands of projects which are currently not sharing any infrastructure; (3) building IT for developing regions is economically viable, in

that there is a market of 4 billion people, but IT must create income for its users (e.g., by offering a franchise model akin to that used to provide cell phone service to rural Bangladesh) because users do not have disposable income; (4) the time is right with the availability of $5 802.11 chipsets, systems on a chip, low-power designs, and solar panels; and (5) this work can have a big impact by reducing poverty and disease and improving the environment, by providing developing regions with a source of income that they can in turn use to improve their standard of living, stability, and security. In terms of concrete systems research areas, Brewer suggested wireless networks, spoken user interfaces for regions with low literacy rates, screens, proxies, and sensors. He indicated that he is starting to work on the problem of IT research for developing countries, and he urged the systems community to consider it to be a part of their research agenda.

George Candea discussed why he believes that wide-area decentralized systems such as peer-to-peer networks are "a good idea whose time has passed." He argued that such systems are hard to build, test, debug, deploy, and manage, and that they have little economic incentive beyond "lack-of-accountability" applications. He suggested that the principles learned from building wide-area distributed systems, e.g., strong componentization, using open protocols, loose coupling, reducing correlated faults through diversity, and writing components while keeping emergent behaviors in mind should be used to build highly dependable "distributed" systems within the datacenter. He summarized by saying, "Don't distribute centralizable apps into the wide-area network (WAN)—take the good ideas from distributed systems and apply them in the system-area network (SAN)."

Geoff Voelker parodied the evolution of data prediction schemes from value prediction (cache previously loaded value and speculatively execute on next load) to "result prediction" (don't bother running the program, just guess the results). He presented several possible applications of result prediction: 5% of the time have gzip just exit with "not in gzip format"; occasionally have the Web browser return random data from its cache, forcing the user to hit reload; occasionally tell the user that he or she has new mail with the "African bank spam"; and—the killer application for result prediction—have SETI@HOME predict no ETs.

Emmett Witchel argued for a "thin malloc" that does not spread data around the address space, but instead keeps it compacted into the smallest memory region possible.

Ethan Miller proposed SCUBA: Scalable Computing for Underwater Biota Assessment, in which 802.11 networks would be deployed on coral reefs.

Sameer Ajmani suggested that systems researchers consider work in computational biology: "We help biologists, then they help thousands of people through pharmaceuticals, genetics, etc.—it's easier than sending computers to Africa." As specific examples of computational biology problems that can directly apply well-known computer science algorithms, he cited string alignment (dynamic programming) and database searches for genes (hash table with 2-tuples and 3-tuples). Andrew Hume added that the National Institutes of Health also has more money than does the National Science Foundation.

Armando Fox called for a bet as to whether a peer-to-peer application would exist before the next HotOS, that would make more sense (economically and technically) to deploy as a peer-to-peer system than as a centralized service. 7 people in the audience said yes, 17 said no. Armando offered to bet someone (Armando taking the "no" side) for a case of alcohol valued less than the conference registration fee in 2005, but there were no takers.

5.2 Scribe Notes by Matt Welsh

In classic HotOS tradition, the Outrageous Opinions session consisted of a stream of short presentations, some serious, some mundane, some hilarious. Jeff Mogul tempted potential participants with a "local delicacy" with a "value of up to $1000" concealed in a paper bag. The level of disappointment in the room was palpable when it was revealed to be a can of Spam.

Val Henson argued against the use of checksums at all levels in a storage system versus end-to-end checksums at the application. Andrew Hume countered that it's good to have accountability at each level when something goes wrong in the system.

Matt Welsh presented a brief but accurate history of systems research goals, culminating in the current trend towards supporting faster downloads of porn and pirating of MP3 files. He argued that systems researchers should be focusing on social problems, such as education and healthcare in developing countries.

Mendel Rosenblum contrasted two approaches to running two untrusted environments on the same hardware—microkernels and VMMs. He wondered aloud who would want to build a microkernel that doesn't leverage existing software, then answered his own question with two possibilities: academics who want to publish papers, or corporate vendors who want to exercise a "control point." Mike Jones asked who these vendors were that were pushing new microkernels, and Mendel responded that Microsoft's Palladium system would be built this way. Much more was said on Palladium in Tuesday's evening session.

Mike Chen argued the opposite of Val Henson's

talk, claiming that one should always use compare by hash, since in compare-by-data the error rate is proportional to the data size and is much larger than the hash collision rate. His real talk, though, was about the Berkeley/Stanford Recovery Oriented Computing project, which aims to improve MTTR in systems. He raised two ways to do this: first, when something fails, redirect the user's attention to something else ("read our NEW privacy policy"), or simply blame it on the user. This buys you tens of seconds to several minutes of wait time.

Timothy Roscoe railed against the construction of scalable systems, claiming that systems should only scale as far as needed and no further. For example, email does not need to scale globally—after all, who needs to send an email message to everybody? After all, whitelisting your email to reduce spam doesn't scale, but it works. Mothy used Google as an example of a system where extra scalability only concentrates power. He cited libraries as a non-scalable alternative to Google.

Dan Wallach talked about all of the recent work on virtualizing resources and getting multiple virtual machines to share resources, such as shared libraries or the OS kernel. He proposed an alternative to these approaches, a radical concept called a "process."

Eric Brewer brought the proceedings back to a more serious note with a call for IT research for developing regions, claiming that we need a direct approach rather than hand-me-downs from previous technology. For these applications we need systems with very low cost and power that make few assumptions about user knowledge or literacy. He pointed out that there are thousands of ongoing IT projects in developing countries but with little sharing. Developing an infrastructure that all these projects could use would be very helpful. Lots of research directions here, including very low-power and low-cost wireless communications, new speech-based user interfaces, network proxies, and sensors.

George Candea argued that WAN P2P distributed systems are a good idea whose time has passed. They are hard to build, test, debug, deploy, and manage, and they offer little economic incentive. He argued that the good distributed systems ideas should be used within the data center instead.

Geoff Voelker presented a novel idea based on the notion of value prediction from hardware architecture—"result prediction." Rather than running the program, we can simply guess the results, leading to excellent speedup potential! Examples: 5% of the time, gunzip could spontaneously report, "Not in gzip format." Web browsers could simply return random stuff from the browser cache. When retrieving new mail, your inbox could simply predict that it's an African Bank spam. The killer app for this technique is clear—SETI@Home.

Emmett Witchel asked whether there is a use for anti-optimization. One application he suggested was to allow users to specify in advance the amount of resources a computation takes, possibly eliminating covert channels. This would be accomplished by intentionally adding delay loops to code to use all available CPU and by spreading allocated memory all over the address space.

More important, Emmett suggested that the last session of the conference be moved forward to replace the breakout session, so that those who had early flights on Wednesday could attend. Most agreed.

Ethan Miller expressed concern about the environment, particularly the disappearance of many species of fish, since we're eating them all. He proposed a sensor-based approach to tracking fish populations, called Scalable Computing for Underwater Biota Assessment. This project can't use 802.11, as it would boil the fish. The main value of this project would be the ability to check the box on the grant proposal form explaining that SCUBA diving is being used in the project.

Sameer Ajmani upheld Matt and Eric's calls for socially relevant applications, pointing out that computational biology has the potential to help thousands of people, and there are real systems problems here (such as managing enormous databases).

And the SPAM award goes to Geoff Voelker!

6. When Things Go Wrong

Chair: Dan Wallach
Scribe: Amit Purohit

6.1 Crash-Only Software

George Candea and Armando Fox, *Stanford University*

The session began with a talk from George Candea in which he explained how to build Internet services that can be safely and cheaply recovered by crash-rebooting minimal subsets of components. He pointed out that most downtime-causing bugs are transient and intermittent, and it is not feasible to guarantee that an application can never crash. Even with recovery-safe applications, recovery can take too long. Crash-only software achieves crash safety and fast recovery by putting all important nonvolatile state outside the application components, into crash-only state stores. For components to be crash-only, they must be decoupled from each other, from the resources they use, and from the requests they process. He conceded that steady-state

performance of crash-only systems may suffer, but argued that (a) the overall goal is to maximize the number of requests successfully served, not to serve some quickly and then be unavailable for a long time, and (b) that techniques will evolve that will improve performance of crash-only systems, the way compilers improved the performance of programs written in high-level languages.

6.1.1 Discussion

Andrew Hume asked how to set timeouts for crashing the systems deliberately. George said that timeouts are set in application-specific ways, at the entrance to the system (e.g., in the Web tier, based on URL). Certain known thresholds can be exploited (e.g., respond within the TCP timeout, or before the HCI-established 8-second distraction threshold for humans). Margo Seltzer asked George to compare their work with that of Stonebraker. George replied that Stonebraker's pioneering ideas were geared mostly toward data management, while their work extends this to recovery management of Internet services. Somebody from Colorado was concerned about the possibility of Do+S attacks by maliciously exploiting this technique. George answered that a stall proxy holds up new requests during recovery; this form of admission control allows the system to quiesce.

6.2 The Phoenix Recovery System: Rebuilding from the Ashes of an Internet Catastrophe

Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker, ***University of California, San Diego***

This presentation explains the design of an operative, distributed, remote backup system called Phoenix. Operating systems and user applications have vulnerabilities. Large numbers of hosts may share vulnerabilities, resulting in major outbreaks. Phoenix uses a strategy based on attributes and cores. By replicating data on a set of hosts with different values for each attribute, it is possible to reduce the probability of error to a negligible number. In the Phoenix system there is no single point of failure; copying with a large number of requests is achieved by exponential backoff.

6.2.1 Discussion

Somebody from MIT was concerned about Phoenix's flexibility. The answer was that the system offers sufficient flexibility to allow different failures. Popescu from Vrije University was concerned about the backup storage, as a Linux machine will always store 10 times more than a Windows machine.

6.3 Using Runtime Paths for Macroanalysis

Mike Chen, ***University of California, Berkeley***; **Emre Kiciman,** ***Stanford University***; **Anthony Accardi,** ***Tellme Networks***; **Armando Fox,** ***Stanford University***; **Eric Brewer,** ***University of California, Berkeley***

The authors emphasized the benefits of micro-analysis, namely latency profiling, failure handling, and detection diagnosis. They introduced the concept of runtime path analysis, in which paths are traced through software components and then aggregated to understand global system behavior via statistical inference. Runtime paths are also used for failure handling and for diagnosing problems, all in an application-generic fashion. The group explained that their work could be extended to p2p message paths, event-driven systems, forks, and joins.

6.3.1 Discussion

To Margo's query about the future of this approach, the authors answered that similar techniques are being used by designers for big projects, thus promising stability.

6.4 Magpie: Online Modelling and Performance-aware Systems

Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan, ***Microsoft Research Ltd, Cambridge, UK***

In this work, the author explains a modeling service that collates traces from multiple machines, extracts request-specific audit trails, and constructs probabilistic models of request behavior. He mentions that workload description and hardware model could be used to predict performance. It is possible to augment the system by adding feedback from past models. To understand the behavior under all scenarios and use the knowledge, behavioral clustering is employed.

A researcher from MIT asked whether the Markov model could be useful. Paul said that some of the Markov solutions could prove useful. Somebody from Washington asked Paul to prove that an unusual activity is always a fault and not a genuine activity. Paul said detection of unusual activity is the basis of intrusion detection systems.

6.5 Using Computers to Diagnose Computer Problems

Joshua A. Redstone, Michael M. Swift, and Brian N. Bershad, ***University of Washington***

Joshua talked about building a global-scale automated problem diagnosis system that captures the

workflow of system diagnosis and repair. The computer generates search terms and locates problem reports. It stores problem reports in a canonical global database. When a problem occurs, the computer detects symptoms and searches the problem database. Joshua argues that more effort building the database could be a good trade-off, resulting in cheaper diagnoses. He said the main challenge lies in creating a database structure so that searching meets user expectations.

6.5.1 Discussion

Mike Jones commented that Windows XP has an automated bug-reporting facility. Joshua said that the XP facility only reports application crashes. Timothy Roscoe said that the main challenge lies in the development of a formal language. Joshua addressed that concern by saying that this is a simple structure. But he agreed that it will be difficult to design the system if problems are disjoint, but he argued that that would not always be the case. Val Henson said Google works perfectly fine for identifying error messages. Joshua argued that his system can give more information, such as the version, which will be more helpful in solving the problem.

6.6 General Discussion

In the general discussion Armando Fox asked a question regarding the deployment-of-diversity problem. Another question was how this system impacts the time needed to recover. The author said he didn't have the final answer, but it should be feasible without much trouble. Jeff Mogul remained concerned about time to recovery. The author said there aren't critical time-recovery requirements, as it is more important to get the data back. He also pointed out that it is possible to make it faster by adding redundancy. But it is a trade-off between storage and time.

Somebody from MIT asked about the deployment of Phoenix. The author said it seems reasonable if there are enough hosts running diversified OSes. Rive Peter compared the Phoenix approach with the randomized replica approach, and questioned whether, with a small number of attribute values, Phoenix really performs better than replicas. The author said that in the randomized replica approach, it is very hard to implement the recovery solution. Ranjitha from UCSD asked Joshua what would be the motivation for people to fill the database. Joshua said organizations tend to use external support, so people would find the increased ease of resolving their problem worth the effort. Mike Jones proposed an alternative scheme which asks the user for a request and if it is solved, then posts the solution. Joshua was not sure how to maintain the database. Mike said you could save the entire request and response sequence and then process it offline.

Jay Lepreau asked George (Crash-only software) about his project's impact on throughput. George said it's likely to be lower, because of the inherently distributed approach (loose coupling, explicit communication, etc.) to building the system, but with time we will learn how to improve performance, citing as an example going from assembly language to high-level languages. High-level languages enabled a qualitative jump in the types of software written, even if the actual writing of code was slower than in assembly. He also argued that "goodput" (throughput of successfully handled requests) is more important than absolute throughput. There was a follow-up question regarding the throughput of applications that do not fail. George said that it's true nobody needs to recover applications that never fail, but that such apps do not exist.

7. Performance Optimization

Chair: Jeff Mogul, Scribe: Ranjita Bhagwan

7.1 Using Performance Reflection in Systems Software

Robert Fowler and Alan Cox, *Rice University*; Sameh Elnikety and Willy Zwaenepoel, *EPFL*

The main idea of this work is to use application-independent measures such as hardware instrumentation mechanisms and general system statistics to adapt system behavior. Performance indicators such as TLB misses and cache misses can be used to measure overhead, while measures of productivity include bytes sent to a network card, as well as flop rate. Productivity and overhead are used to determine whether the system needs to be tuned. Sameh Elnikety showed how server throttling of mySQL using the TPC-W workload succeeded in keeping the throughput at the maximum level while load increased, whereas, without using reflection, the throughput dropped at higher loads.

There was no Q/A.

7.2 Cassyopia: Compiler Assisted System Optimization

Mohan Rajagopalan and Saumya K. Debray, *University of Arizona*; Matti A. Hiltunen and Richard D. Schlichting, *AT&T Labs–Research*

Cassyopia combines program analysis and OS design to do performance optimizations. Whereas the compiler offers a local perspective on optimizations, the OS provides a more general view. Cassyopia tries to bring about a symbiosis between the OS and the compiler. An example of this symbiosis is system call optimization. Cassyopia profiles system call sequences,

then clusters them using compiler techniques. The clustered system calls are called multi-calls. Mohan described preliminary results that showed that significant performance improvement can be obtained using multi-calls.

7.2.1 Discussion

Margo Seltzer asked how one would do error-checking with clustered system calls. How would you know which system call generated the error? Mohan said that they have a means of determining that.

Constantine Sapuntzakis asked for an example of a case in which multi-calls would be useful. Mohan said that Web servers employ a repetitive system call behavior. Another example would be database systems.

7.3 Cosy: Develop in User-Land, Run in Kernel-Mode

Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok, ***Stony Brook University***

User applications are only allowed restricted access, which causes a lot of crossings of the user-kernel boundary. To prevent this, Amit Purohit proposed a compound system call, named Cosy, using which one can execute a user-level code segment in the kernel. The authors have a modified version of gcc that uses Cosy. Kernel safety is ensured by limiting kernel execution time; x86 segmentation and sandboxing techniques can also be used. The performance benefits of Cosy have been reported at 20–80%.

7.3.1 Discussion

Matt Welsh said that there is enough stuff in the OS already that adding more would not gain much. No performance improvement is justified by the loss of reliability that this technique would introduce. Amit said that, at some level, the OS has to be trusted. If it isn't trustworthy, there are lots of other problems that need to be dealt with.

Andrew Hume said that people from Sun Microsystems have said that code executes best in the kernel. Matt Welsh reiterated his statement that reliability was still more important than performance.

Constantine Sapuntzakis asked where the performance improvement came from. Amit said that the savings came from avoiding context switches and from read and write system calls. Constantine asked if the authors had compared their solution to zero-copy solutions. Amit said that zero-copy solutions are applicable only to certain applications; the Cosy solution is more general.

Ethan Miller asked what operations this is useful for, other than zero-copy. Amit said that zero-copy is the primary use right now, since the project is still in its initial stages. They plan to look at I/O next. Andrew Hume said that context switches were important too. Amit said that using compiler techniques and C, it is possible to exploit new avenues for performance improvement.

7.4 General Discussion

Mike Swift said that combining the last two papers might be one way to proceed: take the compiler techniques from Cassyopia to use for kernel execution. Mohan clarified that they do not plan to move any user-level code into the kernel, since he believes that interpretation in the kernel has high overhead. They primarily wanted to reduce the boundary crossing cost, so the two ideologies are not the same. Amit said that interpretation in the kernel is a bottleneck, and so they use a small interpreter. Checking pointers would cost a lot, and so they are using TLBs. This has some overhead, but it's a one-time check. Mohan brought up the point that the aim of Cassyopia is to apply optimizations that are quite obvious, but have not yet been done.

Ethan Miller said that the TLB overhead in Cosy could be unavoidably high. Amit said that is true, but the savings they are getting more than make up for the TLB overhead.

Margo Seltzer drove the nail home by saying that what matters finally is the kernel-user API. All this work on extensible OSes and moving code across the boundary is probably done because the kernel-user API needs to be revisited. So let's fix the API. Applause.

Andrew Hume said that for the applications he has looked at, apart from zero-copy and I/O, there is not much to be gained by putting code into the kernel. Apart from the stated scenarios, the chances of using a multi-call are small. Sometimes reassurance outweighs performance benefits. Mohan said that they are also looking at smaller devices, such as cell phones. All devices are resource-constrained. In these cases, there is also the issue of energy savings. Eric Brewer asked why, for small devices, you would even want a kernel boundary. Mohan answered that cell phones and iPAQs can now have JVMs running on them. They are not targeting reprogrammable devices.

Matt Welsh asked why one would care so much about performance on an iPAQ.

Timothy Roscoe said that since there are different kinds of devices, there should be different OSes for them. And then the question would be where, if anywhere, to put the kernel boundary on them. Whether there is a generic answer to that question is still unclear.

8. Storage 1

Chair: Armando Fox
Scribe: David Oppenheimer

8.1 Why Can't I Find My Files? New Methods for Automating Attribute Assignment

Craig A. N. Soules and Gregory R. Ganger,
Carnegie Mellon University

Craig Soules described new approaches to automating attribute assignment to files, thereby enabling search and organization tools that leverage attribute-based names. He advocates context analysis to augment existing schemes based on user input or content analysis. Context analysis uses information about system state when the user creates and accesses files, using that state to assign attributes to the files. This is useful because context may be related to the content of the file and may be what a user remembers when searching for a file. The Google search engine has proven the usefulness of context analysis; it chooses attributes for a linked site by examining the text associated with the link, and it considers user actions after a search to decide the user's original intent in the search. However, the kind of information Google's context analysis relies on cannot be applied directly to filesystems. In particular, information such as links between pages does not exist in traditional filesystems, and individual filesystems do not have enough users or enough "hot documents" to make Google-like context statistics useful.

Soules described access-based context analysis, which exploits information about system state when a user accesses a file, and inter-file context analysis, which propagates attributes among related files. The former relies on application assistance or existing user input (e.g., filenames), and the latter relies on observing temporal user access patterns and content similarities and differences between potentially related files and versions of the same file. Based on a trace analysis of usage of a single graduate student's home directory tree over a one-month period, Soules concluded that a combination of the techniques he proposes could be useful for automatically assigning attributes. For example, a Web browser can relate search terms to the document the user ultimately downloads as a result of the search, files created and accessed in a single text-editor session can be considered related, and attributes about documents used as input to a distiller such as LaTeX or an image manipulator program can be distilled for attachment as attributes of the output file. Soules also found that examining temporal relationships between file accesses in the trace successfully grouped many related files.

Soules stated that he will be investigating larger user studies, mechanisms for storing attribute mappings, appropriate user interfaces, and how to identify and take advantage of user context switches, i.e., a user's moving from using one program to another.

8.1.1 Discussion

During the question and answer period, Timothy Roscoe observed that tools such as the old AltaVista Personal Edition can index filesystems reasonably effectively. Andrew Hume pointed out that Soules' system does not address situations in which there are multiple access hierarchies for the same underlying files—the original hierarchy is effectively locked in, making it hard to handle multiple filesystem views simultaneously. Soules responded that the focus of this paper was how to get the keywords, and that he has not yet considered how to present the keywords or present multiple views. Hume also suggested looking into software that produces high-quality indexes for printed books.

8.2 Secure Data Replication over Untrusted Hosts

B.C. Popescu, B. Crispo, and A.S. Tanenbaum,
Vrije Universiteit, Amsterdam, The Netherlands

B. C. Popescu described a system architecture that allows arbitrary queries on data content that is securely replicated on untrusted hosts. This system represents an improvement over systems based on state signing, which can support only semi-static data and predefined queries, and systems based on state machine replication, which require operations to be replicated across multiple machines. In the authors' system, every data item is associated with a public/private key pair. The private key is known only to the content owner, and the public key is known by every client that uses the data. There are four types of servers: master servers, which hold copies of content and are run by the content owner; slave servers, which hold copies of data content but are not controlled by a content owner and thus are not completely trusted; clients, which perform read/write operations on content; and an auditor server, described later. The master servers handle client write requests and lazily propagate updates to slave servers. Master servers also elect one of themselves to serve as an auditor, which performs background checking of computations performed by slaves, taking corrective action when a slave is found to be acting maliciously. Slave servers handle client read requests; they may use stale data to handle requests, but clients are guaranteed that once the time parameter maxLatency has passed since a write was committed by a master, no other client will accept a read that is not dependent on the write. All content in the system is versioned; the content version

of a piece of data is initialized to zero when it is created and is incremented each time the data item is updated.

The key challenge in building this system is to enable clients to feel safe having their queries handled by untrusted slave hosts. This is accomplished probabilistically, by allowing clients at any time to send the same request to a (trusted) master and (untrusted) slave and to compare the results. When a slave returns the result of a read, it attaches a signed "pledge" packet containing a copy of the request, the content version timestamped by the master, and the secure hash of the result computed by the slave. If the slave returns an incorrect answer, the "pledge" packet can be used as proof of the slave's malfeasance. This probabilistic checking mechanism is augmented by an auditing mechanism in which, after a client accepts a result from a slave, it forwards the slave's "pledge" packet to a special auditor server. The auditor server is a trusted server that does not have a slave set but just checks the validity of "pledge" packets by reexecuting the requests and verifying that the secure hash of the result matches the secure hash in the packet. The auditor is expected to lag behind when executing write requests, executing writes only after having audited all the read requests for the content version preceding the write.

8.2.1 Discussion

During the question and answer period, Peter Druschel asked what happens to a slave that has been detected to have cheated. Popescu replied that the action is application-specific; once the slave has been taken out of the system, out-of-band means such as a lawsuit can be employed. Rob von Behren asked why you wouldn't need as many auditors as masters. Popescu responded that auditors should be more efficient than masters, because they don't have to communicate back to clients and they can take advantage of techniques such as query optimization, batching, and result reuse. Rob von Behren suggested that the system might be designed to let auditors be untrusted, with numbers of them used in a Byzantine agreement–type configuration. Popescu agreed that this would be a stronger guarantee, but he pointed out that the only weakness in the system as it stands is that an auditor could collude with a slave.

8.3 Palimpsest: Soft-Capacity Storage for Planetary-Scale Services

Timothy Roscoe, ***Intel Research at Berkeley,***
Steven Hand, ***University of Cambridge Computer Laboratory***

Timothy Roscoe described Palimpsest, a "soft-capacity storage service for planetary-scale applications." Palimpsest is designed to serve as a storage service for ephemeral data from planetary-scale applications running on a shared hosting platform such as PlanetLab, XenoServers, or Denali. Examples of the type of data to be stored are static code and data for services, application logs of various sorts, and ephemeral system state such as checkpoints. Despite the temporary nature of the data, it must be highly available during its desired lifetime, thus making single-node local disk storage unsuitable. Traditional filesystems such as NFS and CIFS provide facilities unnecessary for planetary-scale applications and don't meet service provider requirements such as space management, billing, and security mechanisms that allow users to store their data on a shared infrastructure without having to trust the infrastructure provider. Palimpsest aims to provide high data availability for limited periods of time, data protection and resistance to Denial of Service attacks, flexible cost/reliability/performance tradeoffs, charging mechanisms that make sense for service providers, capacity planning, and simplicity of operation and billing. To achieve these goals it uses soft capacity, congestion-based pricing, and automatic space reclamation.

To write a file, a Palimpsest client erasure-codes the file, encrypts each resulting fragment, and sends each encrypted fragment to a fixed-length FIFO queue at the Distributed Hash Table node corresponding to the hash of the concatenation of the filename and the fragment identifier. To retrieve a file, a client generates a sufficient number of fragment IDs, requests them from the block stores, waits until a sufficient number of the requested fragments have been returned, decrypts and verifies them, and recreates the original file. Because the queues are fixed-length, all files stored in Palimpsest are guaranteed to disappear eventually. To keep a file alive the client periodically refreshes it, and to delete the file the client simply abandons it.

The key to predictable file lifetimes is the time constant (T) associated with each block store. T measures how long it takes a fragment to reach the end of the queue and disappear. Clients piggyback requests for information about T on read and write requests, and block stores provide a recent estimate of T by piggybacking on responses. Palimpsest providers charge per (fixed-length) write transaction. Clients can use information about each block store's T value and fee per write transaction to flexibly trade off data longevity, availability, retrieval latency, and robustness. Clients pay providers using anonymous cash transactions. DoS attacks are discouraged by charging for writes. Providers can perform traffic engineering by advertising values of T that deviate from the true value. Congestion pricing is used to encourage users to attain an efficient write rate.

8.3.1 Discussion

During the question and answer period, David Wetherall asked about the possibility of providing a callback to clients when a block they have stored is about to disappear. Roscoe responded that this would add the complexity of having to know about the blocks that are being stored. Sameer Ajmani observed that a load spike might change T significantly. Roscoe agreed; it is up to clients to factor in their belief as to how much the time constant will change when they store their data. Ethan Miller observed that for enough money, a malicious hacker could make people's data disappear. Roscoe replied that one way around that situation would be to dampen the rate at which the time constant could change, by lirniting the write rate. Benjamin Ling asked what would happen if a provider lied, advertising a larger T than was true. Roscoe responded that third-party auditors can monitor the time constant and provide a *Consumer Reports*-like service to evaluate providers' claims. Finally, Jay Lepreau asked whether nonmalicious traffic might cause temporary load spikes, for example, an intrusion detection service using Palimpsest at the time a distributed worm such as SQL Slammer was unleashed. Roscoe replied that this would be a problem, but that hopefully the Palimpsest pricing scheme could help here.

8.4 General Discussion

During the general question and answer period following the session's talks, Andrew Hume asked Timothy Roscoe whether a simpler scheme than Palimpsest could be used to store ephemeral files just like regular files and cycle them using a generational scheme, eventually deleting the files that graduated from the oldest generation. Roscoe responded that Palimpsest provides an easy charging and pricing mechanism, while standard network filesystems such as NFS do not. Val Henson asked Popescu what he thought of the "High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two" talk, in light of the fact that his system is targeted toward storing data on untrusted hosts. Popescu responded that they're more interested in environments in which hosts are less transient than in standard peer-to-peer networks. Jeff Chase observed that Palimpsest clients have no control over the placement of their data, and he asked Roscoe whether he thought that was significant. Roscoe responded that the ability to select specific nodes on which to store data could be provided by an orthogonal mechanism. Benjamin Ling asked whether widespread adoption of Palimpsest would be hindered by the lack of hard guarantees about data longevity. Roscoe responded that legal contracts akin to service level agreements could be layered on top of Palimpsest to ease users' concerns about the inherent risk of data loss. Furthermore, this is really a futures market: third parties can charge premiums for providing guarantees and taking on the risk of data loss themselves.

9. Trusting Hardware

Chair: Dan Wallach
Scribe: Matt Welsh

This session turned out to be the most controversial of the conference, as two of the three talks discussed the use of secure hardware and systems such as Microsoft's Palladium architecture.

9.1 Certifying Program Execution with Secure Processors

Benjie Chen and Robert Morris, *MIT Laboratory for Computer Science*

Benjie Chen is motivated by potential uses for trusted computing hardware other than digital rights management. Since all PCs may include this hardware in the future, he is interested in exploring the hardware and software design for such systems. His running example was secure remote login, such as from a public terminal at an Internet cafe, where the client machine can attest to the server that it is running unadulterated software (OS, ssh client, etc.) Of course, this does not preclude low-tech attacks such as a keyboard dongle that captures keystrokes, but that is outside the immediate problem domain.

Benjie presented an overview of the Microsoft Palladium, or Next Generation Secure Computing Base, architecture, which uses a secure "Nexus" kernel and a secure chip that maintains the fingerprints of the BIOS, bootloader, and Nexus kernel. A remote login application would send an attestation certificate, generated by Nexus and the secure chip, to the server. The issues here are how to keep the Nexus kernel small and how to verify OS services such as memory paging or the network stack. Some ways to improve Palladium's security and verifiability were discussed, such as using a small microkernel that allows attestation of all OS modules above it, as well as a flexible security boundary (where some, not all, of the OS modules are verified). There is a connection with the XOM secure processor work, which prevents physical attacks on DRAM by storing data in encrypted form in memory and only decrypting it into a physically secure cache. Borrowing some of these ideas, one could run the microkernel within the secure processor, which would authenticate all data transfers to DRAM, and the application, hardware drivers, network stack, etc., could all be encrypted in DRAM.

9.1.1 Discussion

Constantine Sapuntzakis asked whether Nexus was not a microkernel. Benjie answered that Nexus implements many drivers inside, which raises the problem of how to verify them all and still keep the security boundary small. Constantine noted that one difficulty with XOM is that replay attacks are possible, and Benjie agreed.

Mendel Rosenblum said that secure remote login is not a good example application, since it doesn't solve the overall security problem (for example, a camera looking over the user's shoulder can capture keystrokes). Benjie agreed that not everyone believes in this application, but there are other applications that drive their design.

Jay Lepreau argued that the real application for this work is DRM. Benjie said they will continue to support DRM, but they are interested in exploring other applications as well.

Constantine pointed out that XBOX is supposedly secure hardware, but a buffer overrun exploit was recently discovered. Benjie replied that they haven't tried to address this problem—using attestation just tells you that the code you thought was running is indeed running. It also turns out that attestation can't deal with the code being modified at runtime.

9.2 Hardware Works, Software Doesn't: Enforcing Modularity with Mondriaan Memory Protection

Emmett Witchel and Krste Asanovic, *MIT Laboratory for Computer Science*

Emmett proposed efficient, word-level memory protection to replace page- or segment-based protection mechanisms. This is motivated by the use of fine-grained modules in software, with narrow interfaces both in terms of APIs and memory sharing. He argued that safe languages are not the answer, in part because it is difficult to verify the compiler and runtime system. Rather, allowing hardware protection to operate at the word level is much simpler and permits a wide range of sharing scenarios. For example, when loading a new device driver into the kernel, the MMP hardware would be configured to permit the module to access its own code and data, as well as to make calls to other modules and share very fine-grained memory (e.g., part of a larger data structure in the kernel). Some of the challenges involve cross-domain calls through call gates; dealing with the stack (since no single protection domain "owns" the stack); and automatically determining module boundaries through information already present in code, such as symbol import/export information in kernel modules. Other potential uses include elimination of memory copies on system calls, specialized kernel entry points, and optimistic compiler optimizations (e.g., write-protecting an object and running cleanup code if a write fault occurs).

9.2.1 Discussion

Mike Swift asked for clarification of the situation where a cross-module call might have multiple targets, as with dynamic dispatch. Emmett replied that the call gate is on the first instruction of the target routine, not the call site itself.

George Candea asked whether the difference between this technique and software fault isolation is simply performance. Emmett answered that SFI is great if you have a very stylized problem, but it has high overhead and has issues with protecting multiple discontiguous regions of data. Also, SFI is just not fast—for Emmett's tests, Purify yields a slowdown of 10–50x, but with MMP it's less than 10%.

Timothy Roscoe asked about the relationship to the SOSP '99 paper on the EROS system, which had a fast capability system running on normal PCs with revocation. Emmett replied that EROS had a nice design, and that the big difference is the level of indirection and how to make it really efficient.

Eric Brewer asked how this approach relates to Multics. Emmett responded that essentially this is a way to do Multics for multi-GHz processors in a modern context. Mendel Rosenblum noted that the people who designed x86 segmentation paid attention to the Multics design, but what they built was the unusable-to-most-software x86 segmentation hardware. Furthermore, he believed that if they looked at Mondriaan, they would pervert it into something bad. Emmett didn't find this a substantive criticism, believing that Intel engineers are more careful these days.

Jay Lepreau argued that MMP does not solve the larger protection domain issue. For example, a thread may call into another module which ends up spinning in an infinite loop or deadlocking. Emmett responded that this is a not a problem that he was attempting to address; he was focusing on the memory protection issue alone.

9.3 Flexible OS Support and Applications for Trusted Computing

Tal Garfinkel, Mendel Rosenblum, and Dan Boneh, *Stanford University*

Tal's talk returned to the question of using secure hardware for applications other than DRM, and shared much of the motivation and background of Benjie's talk. The core problem with open platforms (as opposed to closed platforms, such as ATMs and cell phones), is that applications can be deployed across a wide range

of existing hardware but it is difficult to manage trust. Tal proposed the use of virtual machine monitors to provide a trusted OS environment. The VMM can run either an "open box" VM (such as a standard OS) or a "closed box" VM (a trusted system).

For example, one closed box VM might be a virtual Playstation game console, which uses attestation to prevent cheating in a multiplayer game. Another potential application is a customized distributed firewall pushed into the VMM to protect the network from the host, e.g., by preventing port scanning or IP spoofing or by enforcing connection rate limits. Tal also discussed applications to reputation systems and third-party computing (a la SETI@Home). He concluded the talk with a review of current efforts in this area, including TCPA, Palladium, and LaGrande.

9.3.1 Discussion

Mike Swift admitted that he is not a convert to the VMM concept, claiming that this approach requires that the OS has no bugs. Tal's response was that attestation tells you what the software stack is, allowing you to have more confidence in the system, but Mike countered that this docs not make it inherently more robust. Mendel raised the point that the idea behind VMMs is not to have one secure OS (for example, so you don't need to use Nexus necessarily), but to allow different people to use different secure OSes. Jay Lepreau didn't buy the argument that VMMs are simpler, and wondered how much code there is in VMware. Mendel explained that for architectures that are virtualizable (which, unfortunately, does not include the x86), you can get away with about 10,000 lines of code, not including device drivers. Mendel would expect future architectures to be more virtualization-friendly.

9.4 General Discussion

The political issues surrounding trusted computing platforms raised a number of interesting—and heated—questions from the audience. Dan Wallach started off by describing the recent XBOX hack, where that system was supposedly trusted hardware. Tal and Andrew Hume countered that there are no guarantees of correctness, just tradeoffs in terms of risk assessment. Mendel suggested that cheating at Quake was not a big concern for industry, so this was not of paramount concern in the XBOX.

Timothy Roscoe was disturbed that the three speakers seemed to be too much in agreement, and raised the question of big protection domains (e.g., VMs) versus tiny protection domains (e.g., Mondriaan memory protection). They didn't take the bait, though—Tal said that these approaches were not mutually exclusive.

Jay Lepreau jumped in at this point and raised the concern that nobody has yet demonstrated an entire (real) operating system based on the microkernel model. He again argued that MPP does not solve the whole domain protection issue, since control flow protection is just as important as memory protection. Emmett admitted that there is some complexity involved, but by making memory protection domains both finer-grained and more explicit, programmers have to think about them more carefully and will document them in the code. Jay argued that chopping up a monolithic system in a "half-assed way" makes it more complex, but Emmett countered that most systems software is already making use of well-defined modules, simply without strong protection between them.

Val Henson returned to the trusted computing discussion, arguing that these platforms are more useful for restricting rights (as with DRM) than for giving us more rights. Benjie argued that the bleak view is that this hardware is going to get pushed out regardless, so we should be considering how to use it for something other than DRM. Tal concurred and said that this work was also about intellectual honesty.

Jay Lepreau argued that Tal and Benjie were really just giving cover to the real commercial driver for this technology, which is DRM. Mike Swift wanted to know where the line between research and the corporate world should be drawn, and whether our community should support this work at all. Tal mentioned again that their work is just bringing more information to the table, but Mike drew an analogy with nuclear weapons research, claiming that expanding knowledge is not the only side effect of this work.

Dirk Grunwald wondered whether calling this "trusted computing" was like the rebranding of "out of order execution" to "dynamic execution"—trusted computing cannot deal with hardware attacks, so consumers may be misled into believing it's safer. Tal pointed out that this is no different from calling a protocol "secure."

Jeff Mogul made the point that technology tends to reinforce existing power structures, and that the issue with trusted computing is not security but, rather, whether the technology reinforces existing power relationships or diffuses them. He summarized, "Are you advocating a system that helps the powerful or that helps the weak?" Eric Brewer claimed that he didn't want "trusted" software on his PC, since it only enforces a particular kind of trust relationship between two parties. He would rather have control over his trust relationships, but trusted computing platforms remove that control. Tal admitted that there is a real concern about using Palladium as an industry control point. Timothy Roscoe wrapped up the session by pointing

out that this discussion had been rather "sterile" and had not touched on any of the issues of politics, lawmaking, or market forces surrounding the DRM and trusted technology debate.

10. Pervasive Computing

Chair: Geoff Voelker
Scribe: Ranjita Bhagwan

10.1 Sensing User Intention and Context for Energy Management

Angela B. Dalton and Carla S. Ellis, *Duke University*

Angela Dalton talked about employing low-power sensors to monitor user behavior and then using this information to reduce system energy consumption. She described a case study called faceoff, which uses a camera to perform image capture. This is followed by face detection, and the information is fed into a control loop that does the system-level energy management by turning off the display. The authors have built a prototype which uses image capture and skin detection. A feasibility study to determine the best-case energy savings faceoff can provide showed that significant savings are possible. Also, an important issue is responsiveness in the system. The paper discusses various ways of increasing responsiveness and describes several optimizations with that objective.

10.1.1 Discussion

Geoff Voelker asked if the authors had considered use of the same techniques for security, such as face recognition rather than face detection. Angela agreed that the system offers other benefits, such as privacy, but they have not yet explored them.

Mike Swift asked how robust skin detection is, and whether the detector can be confused. Angela said that the system should detect different skin colors, but it is not a good way of doing face detection. Ideally, you would use something more than skin detection for this purpose. However, it sufficed for the case study.

10.2 Access Control to Information in Pervasive Computing Environments

Urs Hengartner and Peter Steenkiste, *Carnegie Mellon University*

Locating people in a pervasive computing environment can be done at many levels and many granularities. An access control mechanism is required. However, access control mechanisms for conventional information cannot be used as is for such environments. Urs Hengartner described a people locator service that uses digital certificates for defined location policies. The authors use three design policies: identify information early, design policies at the information level, and exploit information relationships. Their approach is to use a step-by-step model, where validation of a client node is done at every server node. They also use a policy description language and an information description language in their people locator service.

10.2.1 Discussion

Sameer Ajmani brought up the issue of certificate systems being an administrative nightmare. He said that this is going to create lots of privacy issues, and that lots of trust issues are assumed. Most individuals are not going to even look at the policies. In response, Urs said that policy generation is done behind the scenes. The front end can be designed depending on the user. It could be a Web browser for an individual, while it could be something else for service developers.

Andrew Hume said that these systems break the notion of practical obscurity. Urs said that you could filter data so that a single node could not do serious damage.

Margo Seltzer asked if the speaker could compare this access control mechanism to that of the mobile people architecture at Stanford. Urs said he was not aware of the Stanford system.

Tal Garfinkel said that the information in the system would be regulated by many different entities. The speaker responded that there are not going to be that many entities. For example, a university runs the service, and you have to trust them anyway.Tal asked if there is any way to shield information in this scenario. The speaker said this could be done only in a legal way.

10.3 Privacy-Aware Location Sensor Networks

Marco Gruteser, Graham Schelle, Ashish Jain, Rick Han, and Dirk Grunwald, *University of Colorado at Boulder*

Marco Gruteser gave a brief description of a system that uses sensor networks to gather information, while making sure that some degree of anonymity is maintained. Describing the problem, Marco said that sensor networks could be used to identify the precise location of certain entities, which could be undesirable. With an anonymous data collection scheme, you do not need to negotiate a privacy policy and the risk of accidental data disclosures is reduced, since databases no longer store the information. Marco described k-anonymity in database systems and said that the concept could be applied to location information.

10.3.1 Discussion

Ethan Miller asked how this interacts with finer-

grained data than location information. Would k-anonymity still be maintained? Marco said that the current work focuses on location information at this granularity. If you look at different pieces of information, the problem becomes more complex.

Margo Seltzer asked whether k-anonymity can be preserved when you aggregate location information obtained from different sensor networks. Marco said that with multiple sensor networks, maintaining k-anonymity could be a problem.

Andrew Hume said that k-anonymity is not sufficient to preserve anonymity. He suggested that Marco consult statisticians about a better means for doing this. Andrew also asked, for reasonable values of k, what applications would this have? Marco responded that for k=10, accuracy is on the order of 100 meters. For many location-based services, that would be sufficient, for instance, detecting whether a meeting room is busy.

Ethan Miller asked how one decides a good value of k. Marco said that it depends on user preferences and on circumstances.

Sameer Ajmani brought up a problem: where one trusts sensor networks, for example, in an office, one usually does not care about anonymity. On the other hand, outside the office, one would not trust a sensor network. Marco suggested a certification process, in which you trust the company building the sensor network, or you trust the government using the sensor network. Essentially, you need to trust the people who deploy the sensor network.

Bogdan Popescu asked whether the sensors need to talk to each other. Marco said yes, they assume a standard multi-mode wireless sensor network, such as Berkeley motes.

10.4 General Discussion

Urs asked Angela whether, when there are multiple windows open and only one active, has she thought about turning off only parts of the screen? Angela said that there is related work that deals with energy adaptive displays. Presently this cannot be done. But you could use some form of gaze tracking to do this. Andrew Hume asked whether the partial screen turn-off works at the hardware level. Angela responded that currently there is no hardware that does that. New kinds of displays are assumed. But you would still need to control it through the system.

Andrew Hume commented that this could be used for covert operations. For example, if a laptop screen shuts down, it means that no one is close to it. Is there a security aspect to that, because you can detect location to some extent? Angela said that larger networks could do this, and, yes, there are lots of security implications.

Val Henson asked what the trend is for built-in sensors. Angela replied that most devices can now have built-in cameras. In general, these sensors are low-power, are cheap, and are becoming pervasive, especially the cameras.

Andrew Hume asked what kind of resolution of the camera is needed to make this work well. Angela said that detection currently is skin-color-based, and you could do this even with a low-resolution black-and-white camera.

Geoff Voelker asked Urs if he had thought of foreign users coming into an administrative domain. Urs responded that since they use SPKI/SDSI digital certificates, they do not require a PKI and could give access to anyone.

Val Henson asked Marco how you decide the shape of the location. Marco said that the current assumption is that the sensors are preconfigured in a room with a preset room ID; the same applies to a building, and so on.

Jay Lepreau asked Angela whether she had data on how people use laptops, so that she could evaluate how well her scheme would work with user behavior patterns. Angela said that more and more people are using laptops as their primary system. Moreover, energy awareness is important generically. However, this question is valid for cell phones, PDAs, etc., which could have widely varying usage patterns. Jay said that the power management on his laptop is frustrating, because it is stupid. It would be good to have a diagnostic tool, with the user being able to guide the system to some extent. Has Angela considered providing the user with a diagnostic tool? Angela said that you could imagine a user interface, which could be used to measure the annoyance factor of the power management, and she agreed that things kicking in at the wrong time can cause problems.

11. Storage 2

Chair: Margo Seltzer
Scribe: David Oppenheimer

11.1 FAB: Enterprise Storage Systems on a Shoestring

Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch, *Hewlett Packard Laboratories*

Alistair Veitch of HP Labs presented "FAB: Federated Array of Bricks." He described a project that is aimed at making a set of low-cost storage bricks behave, in terms of reliability and performance, like an enterprise disk array, but at lower cost and with greater

scalability. The FAB array is built from bricks each of which consists of a CPU, memory, NVRAM, a RAID-5 array of 12 disks, and network cards. Clients connect to the array using a standard protocol such as iSCSI, Fibre Channel, or SCSI, and the storage bricks communicate among themselves using a special FAB protocol running on Gigabit Ethernet. The requirements of the array are zero data loss, continuous availability, competitive performance, scalable performance and capacity, management as a single entity, online upgrade and replacement of all components, and higher-level features such as efficient snapshots and cloning. The principal research challenges are failure tolerance without losing data or delaying clients, asynchronous coordination that does not rely on timely responses from disks or the operating system, and the ability to maximize performance and availability in the face of heterogeneous hardware. The techniques FAB incorporates to achieve these goals are a quorum-based replication scheme, dynamic load balancing, and online reconfiguration.

After outlining FAB's goals and the high-level techniques used to achieve those goals, Veitch described the quorum-based replication scheme used to achieve reliability. It uses at least three replicas for each piece of data, and reads and writes a majority of replicas. It survives arbitrary sequences of failures, achieves fast recovery, and can be lazy about failure detection and recovery, as opposed to needing to do explicit failover. The array configuration that the designers envision achieves a mean time to data loss of about a million years, which Veitch described as "at the bottom end of what's acceptable."

11.1.1 Discussion

George Candea asked whether Veitch could estimate the cost savings of FAB over traditional enterprise disk arrays. Veitch estimated a 20–30% cost savings but emphasized that it depends on the development time and cost. Dan Wallach asked whether today's enterprise disk arrays are truly expensive to develop and produce, or whether the prices of arrays are really just a huge markup over the prices of disks. Veitch replied that disks really are expensive, especially the kinds used in enterprise arrays, such as dual-ported Fibre Channel disks. He added that customized hardware ASICs and integration with firmware contribute to the six-year development cycle between product inception and delivery, requiring work by hundreds of people. FAB is trying to save money by doing everything possible in software.

11.2 The Case for a Session State Storage Layer

Benjamin C. Ling and Armando Fox, *Stanford University*

Benjamin Ling presented "SSM, a recovery-friendly, self-managing session state store." SSM is specialized for storing session state typically associated with user interactions with e-commerce systems. The system assumes a single user making serial access to semi-persistent data. Ling explained that existing solutions such as filesystems, databases, and in-memory replication exhibit poor failure behavior or recovery performance, or both, and that they are difficult to administer and tune. SSM is designed to be recovery-friendly, that is, it can recover instantly without data loss, and it is self-managing in terms of handling overload and performance heterogeneity of components.

SSM is based on a redundant, in-memory hash table distributed across nodes, called bricks, and stateless client stubs, linked with application servers, that communicate with the bricks. Bricks consist of RAM, CPU, and a network interface (no disk). SSM uses a redundancy scheme similar to, but slightly different from, quorums. On writes, a stub writes to some N random nodes (4, in Ling's example) and waits for the first M of the writes to complete (2, in Ling's example); this scheme is used to avoid performance coupling. The remaining writes are ignored. Reads are issued to a single brick. SSM is recovery-friendly in that no data is lost so long as no more than M – 1 disks in a write quorum fail. Moreover, state is available for reading and writing during a brick failure. SSM is crash-only, using no special-case recovery code: when a brick is added to the system or returns to service after a failure, it is simply started up without any need to run a recovery procedure. SSM is self-managing in that it dynamically discovers the performance capabilities of each brick, as follows. Each stub maintains a count of the maximum allowable inflight requests to each brick, using additive increase to grow this window upon each successful request and multiplicative decrease to shrink the window when a brick operation times out. If an insufficient number of bricks are available for writing, either due to failures or due to a full window, the stub refuses the request issued by the client of the stub, thereby propagating backpressure from bricks to clients.

11.2.1 Discussion

Matt Welsh asked about the capacity of the system, and in particular the expected lifetime of the data to be stored in the system, especially in light of the absence of disks. Ling replied that session state generally lives a few hours to a few days. It is usually about 3KB–200KB per user, and Ling estimated that 10–15 bricks would be able to support 400,000 users. Andrew Hume asked what is done with the N – M requests that are not among the first M writes to complete. Ling explained that these requests are just forgotten, and that

they are handled using a garbage-collection–like mechanism.

11.3 Towards a Semantic-Aware File Store

Zhichen Xu and Magnus Karlsson, *HP Laboratories*; Chunqiang Tang, *University of Rochester*; Christos Karamanolis, *HP Laboratories*

Zhichen Xu described pStore. He explained that storage systems are in some ways an extension to human memory, but that computer-based storage systems have been "dumb" because, unlike humans, they do not associate meanings (semantics) with the stored data. For example, when humans search, they consider abstract properties and relationships among objects, and when they store data, they may group objects into categories or record only the differences between objects. Moreover, humans discover the meanings of objects incrementally. These observations motivate the incorporation of versions, deltas, and dependencies among objects stored in the semantic-aware file store.

The semantic-aware file store is intended to create a framework that captures and uses semantic information for fast searching and retrieval of data; stores data efficiently by using data compression based on semantic relationships of data; provides high performance through semantic data hoarding, data placement, replication, and caching; and enables highly available data sharing by balancing consistency and availability according to the data semantics. The semantic-aware file store uses a generic data model based on RDF to capture semi-structured semantic metadata. The challenges Xu described were how to identify the common semantic relations of interest, finding ways to capture semantic information, how to handle dynamic evolution of semantics, and how to define the tools and APIs that users and applications require.

11.3.1 Discussion

George Candea asked Xu to compare his system to existing products that store and track metadata about files. Xu said the main difference is that those systems store metadata in a database, which has a fixed schema, whereas his system uses semi-structured data and therefore allows evolution of semantics. Val Henson asked whether the semantic-aware file store can do better than Citeseer—if she had a number of research papers in her home directory, would the semantic-aware file store be able to relate them to each other as well as Citeseer does? Xu said the answer depends on the tools used to extract the information from the papers; pStore is a framework into which you could plug any number of data-extraction tools.

11.4 General Discussion

During the discussion that followed the talks in this session, Margo Seltzer asked Veitch whether in ten years HotOS will have a paper explaining why FABs are just as expensive as today's storage arrays. Veitch answered, "I hope not, but who knows?" Andrew Hume asked Veitch whether the FAB design makes it difficult to predict performance, as compared to a disk connected to a single machine, especially in the face of failures due to the large number of potential interactions among bricks in the FAB. Veitch responded that the performance can in fact be modeled as a function of factors such as the overhead of doing a disk I/O, the number of nodes, the desired mean time to data loss, and so on. He added that even today no storage system gives you hard-and-fast performance guarantees and that the more difficult question is how to model overall system reliability and the optimal tradeoffs among design parameters. Rob von Behren asked Veitch whether he had considered using non-RAID disks instead of RAID arrays inside each brick. Veitch responded that by using RAID, the mean time to data loss is controlled by the failure rate of each brick rather than the failure rate of individual disks, thereby increasing reliability. He added that it's very difficult to get good numbers on actual failure rates of system components, so estimating overall reliability is difficult.

High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two

Charles Blake cb@mit.edu
Rodrigo Rodrigues rodrigo@lcs.mit.edu
MIT Laboratory for Computer Science

Abstract

Peer-to-peer storage aims to build large-scale, reliable and available storage from many small-scale unreliable, low-availability distributed hosts. Data redundancy is the key to any data guarantees. However, preserving redundancy in the face of highly dynamic membership is costly. We use a simple resource usage model to measured behavior from the Gnutella file-sharing network to argue that large-scale cooperative storage is limited by likely dynamics and cross-system bandwidth — not by local disk space. We examine some bandwidth optimization strategies like delayed response to failures, admission control, and load-shifting and find that they do not alter the basic problem. We conclude that when redundancy, data scale, and dynamics are all high, the needed cross-system bandwidth is unreasonable.

1 Introduction

Recent systems (CAN, Chord, Pastry, or Tapestry [7, 11, 8, 13]) enable peer-to-peer lookup overlays robust to intermittent participation and scalable to many unreliable nodes with fast membership dynamics. Some papers ([1, 6]) express a hope that, with extra data redundancy, *storage* can inherit scalability and robustness from the underlying lookup procedure. More work still ([5, 9]) *implies* this hope by using robust lookup as a foundation for wide-area storage layers, even though this complicates other desirable properties (e.g., server selection).

This paper argues that trying to achieve all three things — scalability, storage guarantees, and resilience to highly dynamic membership — overreaches bandwidth resources likely to be available, regardless of lookup. Our argument is roughly as follows. Simple considerations and current hardware deployment suggest that idle upstream bandwidth is the limiting resource that volunteers contribute, not idle disk space. Further, since disk space grows much faster than access point bandwidth, bandwidth is likely to become even more scarce relative to disk space.

We elaborate this argument in the next section using a generic resource usage model to estimate conservatively the costs associated with maintaining redundancy in systems built from unreliable parts. Section 3 adapts our model to accommodate hosts which are temporarily unavailable but have not lost their data. Section 4 discusses other issues such as admission control or load-shifting, hardware trends, and the importance of incentives. Along the way we use numbers from Gnutella, a real peer-to-peer system, to highlight how bandwidth contributions are the serious limit to scaling data. We conclude in Section 5.

2 A Simple Model

In this section we consider the bandwidth necessary for reliable peer-to-peer storage. We present a simple analytic model for bandwidth usage that attempts to provide broad intuition and still apply in some approximation to currently proposed systems.

2.1 Assumptions

We assume a simple redundancy maintenance algorithm: whenever a node leaves or joins the system, the data that node either held or will hold must be downloaded from somewhere. Note that by *join* and *leave* we mean really joining the system for the first time or leaving forever. We do not refer to transient failures, but rather the intentional or accidental loss of the contributed data. Section 3 elaborates this model to account for temporary disconnections that may not trigger data transfers. We also assume there is a static data placement strategy (i.e., a function from the current membership to the set of replicas of each block).

We make a number of simplifying assumptions. Each one is *conservative* — increased realism would increase the bandwidth required. Note that any storage guarantee effectively insists that the probability of not getting a datum is below some threshold. The time to create new nodes must therefore consider the worst-case accidents of data distribution and other variations. Therefore, the fact that we perform an average case analysis makes our model conservative.

We assume identical per-node space and bandwidth contributions. In reality, nodes may store different amounts of data and have different bandwidth capabilities. Maintaining redundancy may require in cer-

tain cases more bandwidth than the average bandwidth. Creating more capable nodes from a set of less capable nodes might take more time. Average space and bandwidth therefore conservatively bound the worst case which is the relevant bound for a guarantee.

We assume a constant rate of joining and leaving. As with resource contributions, the worst case is a more appropriate figure to use for any probabilistic bound. The average rate bounds the maximum rate from below, which is again conservative. We also assume independence of leave events. Since failures of networks and machines are not truly independent, more redundancy would really be required to provide truer guarantees.

We assume a constant steady-state number of nodes and total data size. A decreasing population requires more bandwidth while an increasing one cannot be sustained indefinitely. It would also be more realistic to assume data increases with time or changes which would again require more bandwidth.

2.2 Data Maintenance Model

Consider a set of N identical hosts which cooperatively provide guaranteed storage over the network. Nodes are added to the set at rate α and leave at rate λ, but the average system size is constant, i.e. $\alpha = \lambda$. On average, a node stays a member for $T = N/\lambda$.

Our data model is that the system reliably stores a total of D bytes of unique data stored with a redundancy expansion factor k, for a total of $S = kD$ bytes of contributed storage. One may think of k as either the replication factor or the expansion due to coding. The desired value of k depends on both the storage guarantees and redundant encoding scheme and is discussed more in the next section.

We now consider the data maintenance bandwidth required to maintain this redundancy in the presence of a dynamic membership. Note that the model does not consider the bandwidth consumed by queries, and therefore we present a conservative bandwidth estimate.

Each node *joining* the overlay must download all the data which it must later serve, however that subset of data might be mapped to it. The average size of this transfer is S/N. Join events happen every $1/\alpha$ time units. So the aggregate bandwidth to deal with nodes joining the overlay is $\frac{\alpha S}{N}$, or S/T.

When a node *leaves* the overlay, all the data it housed must be copied over to new nodes, otherwise redundancy would be lost. Thus, each leave event also leads to the transfer of S/N bytes of data. Leaves therefore also require an aggregate bandwidth of $\frac{\lambda S}{N}$, or S/T. The total bandwidth usage for all data maintenance is then $\frac{2S}{T}$, or a per node average of:

$$B/N = 2\frac{S/N}{T}, \text{ or } BW/node = 2\frac{space/node}{lifetime} \quad (1)$$

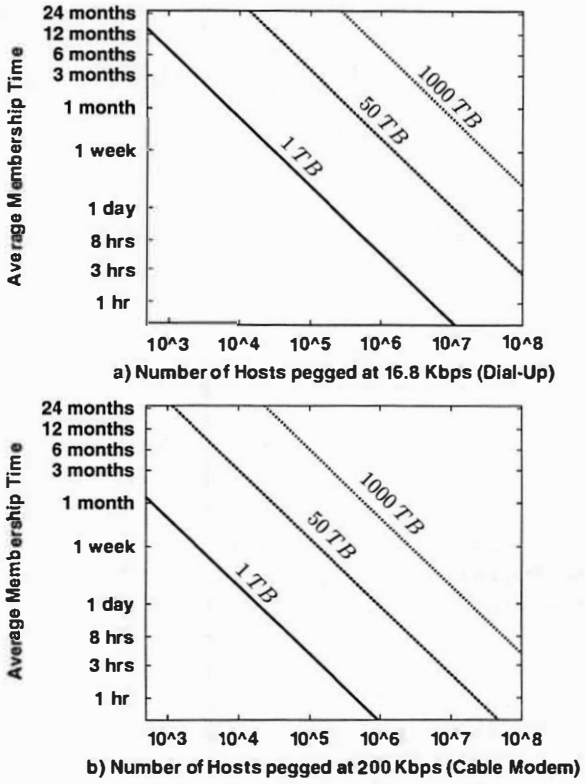


Figure 1. Log-Log plots for the participation requirements of a) dial-up and b) cable modem networks. Plotted are thresholds below which various amounts of unique data will incur over 50% link saturation just to maintain the data. These use a redundancy $k = 20$.

2.3 Understanding the Scaling

Figure 1 plots some example "threshold curves" in the lifetime-membership plane. This is the basic participation space of the system. More popular systems will have more hosts, and those hosts will stay members longer. Points below a line for a particular data scale require data maintenance bandwidth in excess of the available bandwidth. We plot thresholds for maintenance *alone* consuming half the total link capacity for dial-ups and cable modems. The data scales we chose, 1 TB, 50 TB, and 1000 TB, might very roughly correspond to a medium-sized music archive, a large music archive, and a small video archive (a few thousand movies), respectively.

There are two basic points to take away from these plots. First, short membership times create a need for enormous node counts to support interesting data scales. E.g., a million cable modem users must each provide a *continuous month* of service to maintain 1000 TB even if no one ever actually queries the data! Second, this strongly impacts how fast the storage of such a network

can grow. At a monthly turnover rate, each cable modem must contribute less than 1 GB of unique data, or 20 GB of total storage. Given that PCs last only a few years and a few years ago 80 GB disks were standard on new PCs, 20 GB is likely about or below current idle capacity.

Figure 1 uses a fixed redundancy factor $k = 20$. The actual redundancy necessary depends on T, N, probability targets for data loss or availability. Section 3 examines in more detail the necessary k for both replication-style and erasure coded redundancy for availability.

3 Availability and Redundancy

This section expands our model to include hosts that are transiently disconnected and estimates redundancy requirements in more detail.

3.1 Downtime vs. Departure

So far our calculations have assumed that the resources a host contributes are always available. Real hosts vary greatly in availability [3, 4, 10]. The previous section shows that it takes a lot of bandwidth to preserve redundancy upon departures. So it helps to distinguish true departures from temporary downtime, as in [3].

Our model for how systems distinguish true departures from transient failures is a membership timeout, τ, that measures how long the system delays its response to failures. I.e., the process of making new hosts responsible for a host's data does not begin until that host has been out of contact for longer than time τ.

Counting offline hosts as members has two consequences. First, member lifetimes are longer since transient failures are not considered leaves. Second, hosts serve data for a *fraction* of the time that they are members (or a fraction of members serve data at a given moment). We define this fraction to be the availability, a.

Since only a fraction of the members serve data at a time, more redundancy is needed to achieve the same level of availability. Also, the effective bandwidth contributed per node is reduced since these nodes serve only a fraction of the time. Thus, the membership lifetime benefits gained by delayed response to failures are offset by the need for increased redundancy and reduced effective bandwidth. To understand this effect more quantitatively we must first know the needed redundancy.

3.2 Needed Redundancy: Replication

First we compute the data expansion needed for high availability in the context of replication-style redundancy. Note that availability implies reliability since lost data is inherently unavailable.

Average lifetime now depends on timeout: $T = T_\tau$. System size and availability also depend on τ, and $N_\tau = N_0/a_\tau$, by our definition of availability.

We wish to know the replication factor, k_a, needed to achieve some per object unavailability target, ϵ_a. (I.e., 1 ϵ_a has some "number of 9s".)

$$\begin{aligned}\epsilon_a &= P(\text{object } o \text{ is unavailable}) \\ &= P(\text{all } k_a \text{ replicas of } o \text{ are unavailable}) \\ &= P(\text{one replica is unavailable})^{k_a} \\ &= (1 \quad a_\tau)^{k_a}\end{aligned}$$

which upon solving for k_a yields

$$k_a = \frac{\log \epsilon_a}{\log(1 \quad a_\tau)} \approx \frac{\log 1/\epsilon_a}{a_\tau} + O(a^2) \qquad (2)$$

We can now evaluate the tradeoff between data maintenance bandwidth and membership timeout. We account for partial availability by replacing B with $a_\tau B$ in Equation (1). Solving for B/N and substituting Equation (2) gives:

$$B_\tau/N_\tau = \frac{2k_a D}{N_\tau a_\tau T_\tau} = \frac{2D \qquad \log \epsilon_a}{N_\tau a_\tau T_\tau \log(1 \quad a_\tau)} \qquad (3)$$

To apply Equation (3) we must know N_τ, a_τ, and T_τ, which all depend upon participant behavior. We estimate these parameters using data we collected in a measurement study of the availability of hosts in the Gnutella file sharing network. We used a methodology similar to a previous study [10], except that we allowed our crawler to extract the entire membership, therefore giving us a precise estimate of N_τ. Our measurements took place between April 11, 2003 and April 19, 2003.

Figure 2 suggests that discriminating downtime from departure can lead to a factor of 30 savings in maintenance bandwidth. It seems hopeless to field even 1 TB at high availability with Gnutella-like participation.

3.3 Needed Redundancy: Erasure Coding

A technique that has been proposed by several systems is the use of erasure coding [12, 2]. This is more efficient than conventional replication since the increased intra-object redundancy allows the same level of availability to be achieved with much smaller additional redundancy. We now exhibit the analogue of Equation (2) for the case of erasure coding.

With an erasure-coded redundancy scheme, each object is divided into b blocks which are then stored with an effective redundancy factor k_c. The object can be reconstructed from any available m blocks taken from the stored set of $k_c b$ blocks (where $m \approx b$). Object availability is given by the probability that at least b out of $k_c b$ blocks are available:

$$1 \quad \epsilon_a = \sum_{i=b}^{k_c b} \binom{i}{k_c b} a^i (1 \quad a)^{k_c b \;\; i}.$$

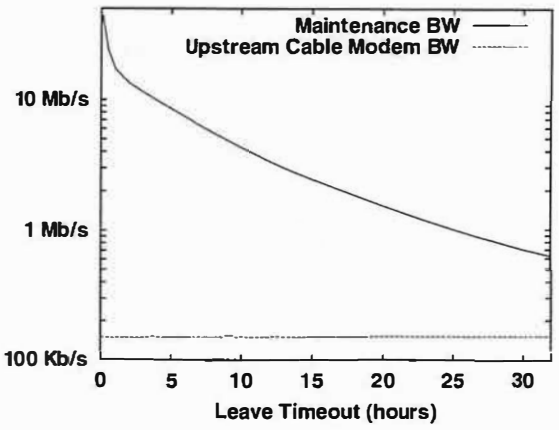


Figure 2. Per node bandwidth to maintain 1 TB of unique data at 6 nines of per-object availability with the system dynamics of 33,000 Gnutella hosts. Bandwidth is lessened by longer delays responding to failures, but remains quite large in terms of home Internet users. Each host contributes only about 3 GB.

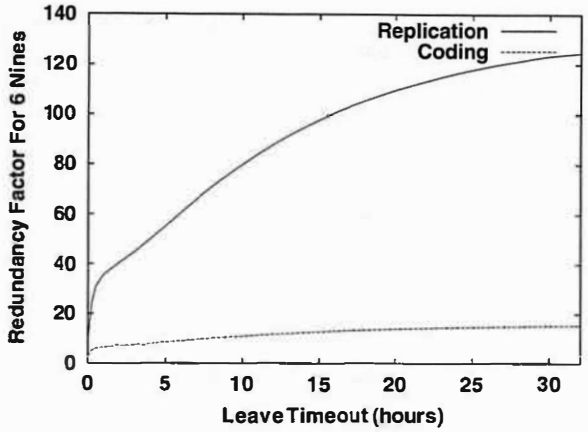


Figure 3. This graph shows that decreased availability from delayed response to failure causes a marked increase in the necessary redundancy. While coding beats replication, the bandwidth savings are only a factor of 8 for our Gnutella trace.

Using algebraic simplifications and the normal approximation to the binomial distribution (see [2]), we get the following formula for the erasure coding redundancy factor and then expand it in a Taylor series:

$$k_c = \left(\frac{\sigma_\epsilon \sqrt{\frac{a(1-a)}{b}} + \sqrt{\frac{\sigma_\epsilon^2 a(1-a)}{b} + 4a}}{2a} \right)^2 \quad (4)$$

$$\approx \frac{\sigma_\epsilon^2}{4b}(1+q)^2 q \quad \tfrac{1}{2}\left(\frac{1}{a} \quad \frac{1}{q} + O(a)\right) \quad (5)$$

$$where \quad q = \sqrt{1 + \frac{4b}{\sigma_\epsilon^2}}.$$

σ_ϵ is the number of standard deviations in a normal distribution for the required level of availability, as in [2]. E.g., $\sigma_\epsilon = 4.7$ corresponds to six nines of availability.

Figure 3 shows the benefits of coding over replication when one uses $b = 15$ fragments. Rather than a replication factor of 120, one can achieve the same availability with only 15 times the storage using erasure codes, for large values of τ, an 8-fold savings. This makes it borderline feasible to store 1 TB of unique data with Gnutella-like participation and about 75 $Kbps$ while-up per node maintenance bandwidth. Utilization is correspondingly *lower* for the same amount of unique data. Only 500 MB of disk per host is contributed. This is surely less than what peers are willing to donate.

Note that all of this is for maintenance only. It would be odd to engineer such highly availabile data and not read it. An actual load is hard to guess, but, as a rule of thumb, one would probably like maintenance to be less than half the total bandwidth. So, the total load one might expect would be greater than 150 $Kbps$ or just at the limit of what cable modems can provide.

This is also not very much service. Only 5,000 of the 33,000 Gnutella hosts were usually available. If all these hosts were cable modems, the aggregate bandwidth available would be about 500 $Mbps$, or 250 $Mbps$ if half that is used for data maintenance. By comparison, the same level of service could be provided by five reliable, dedicated PCs, each with a few $300, 250 GB drives and 50 $Mbps$ connections up 99% of the time.

4 Discussion

This section discusses other issues related to the agenda and design of large-scale peer-to-peer storage.

4.1 Admission Control, Load-Shifting

Another strategy to reduce redundancy maintenance bandwidth is to attempt to not admit highly volatile nodes or very similarly shift responsibility to non-volatile hosts. Fundamentally, this strategy weakens how dynamic and peer-symmetric the network one is envisioning. Indeed, a strong enough bias converts the problem into a garden variety distributed systems problem — building a larger storage from a small number of highly available collaborators.

As Figure 4 shows, in Gnutella, the 5% most available hosts provide 29 of the total 72 service years or 40%. The availability of these 6,000 nodes is about 40% on average. If one is generous, one may also view this 5%-subset of more available hosts as a fairer model of the behavior of a hypothetical population of peer-to-peer participants. We repeated our analysis of earlier sections

using just this subset of hosts with a one day membership timeout. The resulting bandwidth requirement is 30 $Kbps$ per node per unique-TB using coding. Using delayed response, coding, and admission control together enables a 1000-fold savings in maintenance bandwidth over the bleak results at the left edge of Figure 2.

The total scale of this storage remains bounded by bandwidth, though. If the 6,000 best 5% of Gnutella peers each donated 3 GB each then a total of 3 TB could be served with six nines of availability. These hosts would each use 100 $Kbps$ of maintenance bandwidth whenever they were participating. Assuming the query load was also about 100 $Kbps$ per host, cable modems would still be adequate to serve this data. The same service also could be supported by 10 universities, each using $\frac{1}{3}$ of the typical OC3 connections and a $1,500 PC.

Stricter admission control rapidly leads to a subset 967 Gnutella hosts with 99.5% availability. This surpasses even observed enterprise wide behavior [4]. The cost of this improvement is a reduction in service time by 10-fold. The real service reduction will depend on the correlation between availability and servable bandwidth. Ideally, this correlation would be strong and positive.

If the per-node bandwidth of the best hosts is roughly 10-fold the per node bandwidth of the excluded hosts then the total service is only cut in half by using just good nodes. Stated in reverse, leveraging tens of thousands of flaky home users only doubles total data service. This fact is further backed up by a simple back-of-the-envelope calculation. Two million cable modem users at 40% availability can serve about as much bandwidth as 2,000 typical high availability universities allowing half their bandwidth for file sharing.

4.2 Hardware Trends

The discussion so far suggests that even highly optimized systems can achieve only a few GB per host with Gnutella-like hosts and cable-modem like connections. However, hardware trends are unpromising.

Year	Disk	Home access		Academic access	
		Speed (Kbps)	Days to send	Speed (Mbps)	Time to send
1990	60 MB	9.6	0.6	10	48 sec
1995	1 GB	33.6	3	43	3 min
2000	80 GB	128	60	155	1 hour
2005	0.5TB	384	120	622	2 hour

Table 1. Generous bandwidth estimates suggest distributing local disk will get harder. Disk increased by 8000-fold while bandwidth increased only 50-fold.

A simple thought experiment helps us realize the implications of this trend. Imagine how long it would take to upload your hard disk to a friend's machine. Table 1 recalls how this has evolved for "typical" users in recent times. The fourth and sixth columns show an ominous trend for disk space distributors. Disk upload time is getting larger quickly. If peers are to contribute meaningful fractions of their disks their participation must become more and more stable. This supports the main point of this paper: synchronizing randomly distributed, large-scale storage is expensive now, dynamic membership makes it worse, and this situation is worsening.

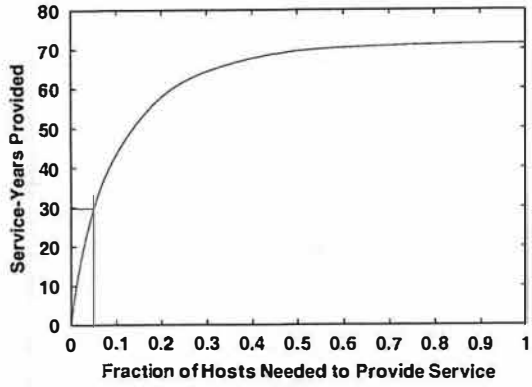


Figure 4. A graph showing how most service time in Gnutella is provided by a tiny fraction of hosts. 5% of hosts provide 40% of the total service time in a three day subset of our trace.

4.3 Incentive Issues

Unlike pioneer systems like Napster and Gnutella, current research trends are toward systems where users serve data that they may have no particular interest in. A good fraction of their outbound traffic might be saturated by access to this data. Storage guarantees exacerbate the problem by inducing a great deal of synchronization traffic above and beyond access traffic. These higher costs may make participation even more capricious than our example of the Gnutella network. Given that stable membership is necessary to reach even modest data scales, participation must be strongly incentivized.

The added value of service guarantees might seem to be one incentive. However, this is not stable since a noticeable downward fluctuation in popularity will make the provided service decline. Another option is having user interfaces which discourage or disallow disconnection, but this is very much against the spirit of a volunteer or donation based system. One reasonable idea is to allow client bandwidth usage to be only proportional to contributed bandwidth. Enforcing this raises many design issues, but since it is an extension to the threat model it seems inappropriate to relegate it to an afterthought.

5 Conclusion

This paper argues that the real scalability problem for robust, Internet-scale storage is not lookup state or procedure, but rather is the *service* bandwidth to field queries and maintain redundancy. For DHT-style systems, maintaining redundancy takes cross-system bandwidth proportional to data scale and membership dynamics. All three properties — redundancy, data, and dynamics — can be high only when cross-system bandwidth is enormous.

The conflict between high availability, large data scales, home user-like bandwidth and fast participation dynamics raises many questions about current DHT research trajectories. In dynamic deployment scenarios, why leverage many nodes to serve data a few reliable ones might? In static deployment scenarios, small lookup-state optimizations may do more harm than good in terms of system complexity and other properties, especially if designers insist on implementing other optimizations in membership state restricted ways. If storage guarantees are inappropriate for large-scale peer-to-peer why worry about lookup guarantees? When anonymity or related security properties are the high priority guarantees, it seems bad to plan on incorporating defenses against threats to these properties as an afterthought.

Acknowledgements

We would like to thank Stefan Saroiu, Krishna Gummadi, and Steven Gribble for supplying the data collected in their study. This research is supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory, NSF Grant IIS-9802066, and a Praxis XXI fellowship.

References

[1] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, pages 43-48, February 2003.

[2] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, UCSD, November 2002.

[3] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.

[4] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[5] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.

[6] John Kubiatowicz. Extracting guarantees from chaos. *Communications of the ACM*, pages 33–38, February 2003.

[7] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM '01 Conference*, San Diego, CA, August 2001.

[8] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-s cale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[9] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.

[10] S. Saroiu, P. K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking 2002 (MMCN'02)*, January 2002.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM '01 Conference*, San Diego, CA, August 2001.

[12] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.

[13] Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

One Hop Lookups for Peer-to-Peer Overlays

Anjali Gupta Barbara Liskov Rodrigo Rodrigues
MIT Laboratory for Computer Science
{anjali,liskov,rodrigo}@lcs.mit.edu

Abstract

Current peer-to-peer lookup algorithms have been designed with the assumption that routing information at each member node must be kept small, so that the bookkeeping required to respond to system membership changes is also small. In this paper, we show that this assumption is unnecessary, and present a technique that maintains complete routing tables at each node. The technique is able to handle frequent membership changes and scales to large systems having more than a million nodes. The resulting peer-to-peer system is robust and can route lookup queries in just one hop, thus enabling applications that cannot tolerate the delay of multi-hop routing.

1 Introduction

Structured peer-to-peer overlay networks like CAN [6], Chord [10], Pastry [8], and Tapestry [11] provide a substrate for building large-scale distributed applications. These overlays allow applications to locate objects stored in the system in a limited number of overlay hops.

Peer-to-peer lookup algorithms strive to maintain a small amount of per-node routing state – typically $O(\log N)$ – because they expect that system membership changes frequently. This expectation has been confirmed for successfully deployed systems. A recent study [9] shows that the average session time in Gnutella is only 2.9 hours. This is equivalent to saying that in a system with 100,000 nodes, there are about 19 membership change events per second.

Maintaining small tables helps keep the amount of bookkeeping required to deal with membership changes small. However, there is a price to pay for having only a small amount of routing state per node: lookups have high latency since each lookup requires contacting several nodes in sequence.

This paper questions the need to keep routing state small. We take the position that maintaining full routing state (i.e., a complete description of system membership) is viable. We present techniques that show that nodes can maintain this information accurately, yet the communication costs are low. The results imply that a peer-to-peer system can route very efficiently even though the system is large and membership is changing rapidly.

We present a novel peer-to-peer lookup system that maintains complete membership information at each node, and show analytic results that prove that the system meets our goals of reasonable accuracy and bandwidth usage. It is, of course, easy to achieve these goals for small systems. Our algorithm is designed to scale to large systems, e.g., systems with more than 10^5 nodes.

The rest of the paper is organized as follows: Section 2 describes the organization of our routing subsystem and Section 3 provides an analysis that shows that the overall cost of maintaining complete routing information is small. Section 4 discusses related work. We conclude with a discussion of what we have accomplished.

2 System Design

We consider a system of n nodes, where n is a large number like 10^5 or 10^6. We assume dynamic membership behavior as in Gnutella, which is representative of an open Internet environment. From the study of Gnutella and Napster [9], we deduce that systems of 10^5 and 10^6 nodes would show around 20 and 200 membership changes per second, respectively. We call this rate r. We refer to membership changes as events in the rest of the paper.

Every node in the overlay is assigned a random 128-bit node identifier. Identifiers are ordered in an *identifier ring* modulo 2^{128}. We assume that identifiers are generated such that the resulting set is uniformly distributed in the identifier space, for example, by setting a node's identifier to be the cryptographic hash of its network address. Every node has a predecessor and a successor in the identifier ring, and it periodically sends keep-alive messages to these nodes. Similarly, we associate a successor node with every 128-bit key *key*; this is the first node in the identifier ring clockwise from *key*. This mapping from keys to nodes is based on the one used in Chord [10], but changing our system to use other mappings is straightforward.

Clients issue queries that try to reach the successor node of a particular identifier. We intend our system to satisfy a large fraction, f, of the queries correctly on the *first* attempt. Our goal is to support high values of f, e.g., $f = 0.99$. A query may fail in its first attempt due to a membership change, if the notification of the change has not reached the querying node. In such a case, the query

can still be rerouted and succeed in a higher number of hops. Nevertheless, we define failed queries as those that are not answered correctly in the *first* attempt, as our objective is a one hop lookup.

To achieve this goal, every node in the system must keep a full routing table containing information about every node in the overlay. The actual value of f depends on the accuracy of this information.

2.1 Membership Changes

To maintain correct full routing tables, a notification of membership change events, i.e., joins and leaves, must reach every node in the system within a specified amount of time (depending on what fraction of failed queries, i.e., f, is deemed acceptable). Our goal is to do this in a way that has reasonable bandwidth consumption (since this is likely to be the scarcest resource in the system) without increasing notification delay.

We achieve this goal by superimposing a well-defined hierarchy on the system. This hierarchy is used to form dissemination trees, which are used to propagate event information.

We impose this hierarchy on a system with dynamic membership by dividing the 128-bit circular identifier space into k equal contiguous intervals called slices. The ith slice contains all nodes currently in the overlay whose node identifiers lie in the range $[i \cdot 2^{128}/k, (i+1) \cdot 2^{128}/k)$. Since nodes have uniformly distributed random identifiers, these slices will have about the same number of nodes at any time. Each slice has a *slice leader*, which is chosen dynamically as the node that is the successor of the mid-point of the slice identifier space. For example, the slice leader of the ith slice is the successor node of the key $(i + 1/2) \cdot 2^{128}/k$. When a new node joins the system it learns about the slice leader from one of its neighbors along with other information like the data it is responsible for and its routing table.

Similarly, each slice is divided into equal-sized intervals called units. Each unit has a *unit leader*, which is dynamically chosen as the successor of the mid-point of the unit identifier space.

Figure 1 depicts how information flows in the system. Whenever a node (labeled **X** in Figure 1) detects a change in membership (its successor failed or it has a new successor), it sends an event notification message to its slice leader (**1**). The slice leader collects all event notifications it receives from its own slice and aggregates them for t_{big} seconds before sending a message to other slice leaders (**2**). To spread out bandwidth utilization, communication with different slice leaders is not synchronized, the slice leader ensures only that it communicates with each individual slice leader once every t_{big} seconds. Therefore, messages to different slice leaders are sent at different points in time and contain different sets of events. The slice leaders aggregate messages they receive for a short time period t_{wait} and then dispatch the aggregate message to all unit leaders of their respective slices (**3**). A unit leader piggybacks this information on its keep-alive messages to its successor and predecessor (**4**). Other nodes propagate this information in one direction: if they receive information from their predecessors, they send it to their successors and vice versa. This information is piggy-backed on keep-alive messages. In this way, all nodes in the system receive notification of all events. Nodes at unit boundaries do not send information to their neighboring nodes outside their unit. This ensures that there is no redundancy in the communications: a node will get information only from its neighbor that is one step closer to its unit leader. This implies that within a unit, information is always flowing from the unit leader to the ends of the unit.

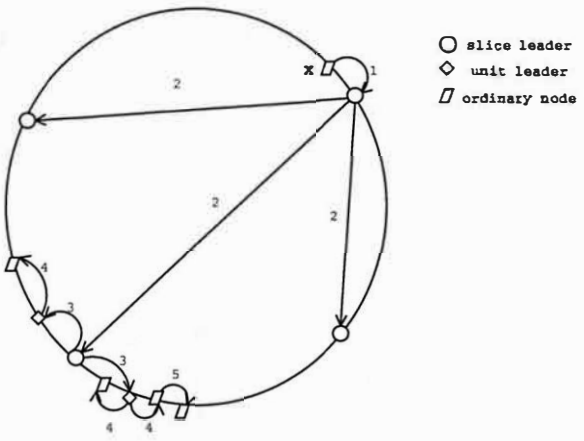


Figure 1: Flow of event notifications in the system

The choice of the number of levels in the hierarchy involves a tradeoff. A large number of levels implies a larger delay in propagating the information, whereas a small number of levels generates a large load at the nodes in the upper levels. We chose a three level hierarchy because it leads to reasonable bandwidth consumption, as we will show in Section 3.

We get several benefits from choosing this design. First, it imposes a structure on the system, with well-defined event dissemination trees. This structure helps us ensure that there is no redundancy in communications, which leads to efficient bandwidth usage.

Second, aggregation of several events into one message allows us to avoid small messages. Small messages represent a problem since the protocol overhead becomes significant relative to the message size, leading to higher bandwidth usage.

2.2 Fault Tolerance

If a query fails on its first attempt it does not return an error to an application. Instead, queries can be rerouted: if a lookup query from node n_1 to node n_2 fails because n_2 is no longer in the system, n_1 can retry the query by sending it to n_2's successor. If the query failed because a recently joined node, n_3, is the new successor for the key that n_1 is looking up, then n_2 can reply with the identity of n_3 (if it knows about n_3), and n_1 can contact it in a second routing step.

Since our scheme is dependent on the correct functioning of unit leaders and slice leaders, we need to recover from their failure. Note that since there are relatively few slice and unit leaders, their failures are less frequent. Therefore, we do not have to be very aggressive about replacing them in order to maintain our query success target. When a slice or unit leader fails, its successor soon detects the failure and becomes the new leader. The successor of a failed unit leader will communicate with its slice leader to obtain recent information. The successor of a failed slice leader will communicate with its unit leaders and other slice leaders to recover information about the missed events.

2.3 Scalability

Slice leaders have more work to do than other nodes, and this might be a problem for a poorly provisioned node with a low bandwidth connection to the Internet. To overcome this problem we can identify well connected and well provisioned nodes as "supernodes" on entry into the system. There can be a parallel ring of supernodes, and the successor (in the supernode ring) of the midpoint of the slice identifier space becomes the slice leader. We do require a sufficient number of supernodes so that we can expect that there are at least a few per slice.

As we will show in Section 3, bandwidth requirements are small enough to make most participants in the system potential supernodes in a 10^5 sized system (slice leaders will require 35 kbps upstream bandwidth). In a million node system we may require supernodes to be well-connected academic or corporate users (the bandwidth requirements increase to 350 kbps).

3 Analysis and Choice of System Parameters

This section presents an analysis of how to parameterize the system to satisfy our goal of fast propagation. To achieve our desired success rate, we will need to propagate information about events within some time period t_{tot}; we show how to compute this quantity in Section 3.1. Yet we also require good performance, especially with respect to bandwidth utilization. Sections 3.2 and 3.3 show how we satisfy this requirement by controlling the number of slices and units.

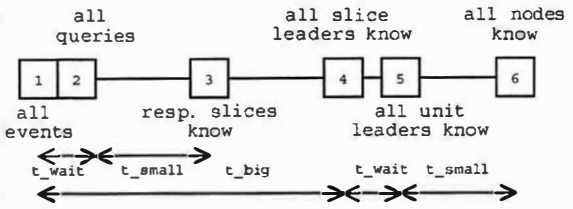


Figure 2: Timeline of the worst case situation

Our analysis considers only non-failure situations. It does not take into account overheads of slice and unit leader failure because these events are rare. It also ignores message loss and delay since this simplifies the presentation, and the overhead introduced by message delays and retransmissions is small compared to other time constants in the system.

Our analysis assumes that query targets are distributed uniformly throughout the ring. It is based on a worst case pattern of events, queries, and notifications: we assume all events happen just after the last slice-leader notifications, and all queries happen immediately after that, so that none of the affected routing table entries has been corrected and *all* queries targeted at those nodes (i.e., the nodes causing the events) fail. In a real deployment, queries would be interleaved with events and notifications, so fewer of them would fail.

This scenario is illustrated by the timeline in Figure 2. Here t_{wait} is the frequency with which slice leaders communicate with their unit leaders, t_{small} is the time it takes to propagate information throughout a unit, and t_{big} is the time a slice leader waits between communications to some other slice leader. Within $t_{wait} + t_{small}$ seconds (point 3), slices in which the events occurred all have correct entries for nodes affected by the respective events. After t_{big} seconds of the events (point 4), slice leaders notify other slice leaders. Within a further $t_{wait} + t_{small}$ seconds (point 6), all nodes in the system receive notification about all events.

Thus, $t_{tot} = t_{detect} + t_{wait} + t_{small} + t_{big}$. The quantity t_{detcct} represents the delay between the time an event occurs and when the leader of that slice first learns about it.

3.1 Configuration Parameters

The following parameters characterize a system deployment:

1. f is the acceptable fraction of queries that fail in the first routing attempt
2. n is the expected number of nodes in the system

3. r is the expected rate of membership changes in the system

Given these parameters, we can compute t_{tot}. Our assumption that query targets are distributed uniformly around the ring implies that the fraction of failed queries is proportional to the expected number of incorrect entries in a querying node's routing table. Given our worst case assumption, all the entries concerning events that occurred in the last t_{tot} seconds are incorrect and therefore the fraction of failed queries is $\frac{r \times t_{tot}}{n}$. Therefore, to ensure that no more than a fraction f of queries fail we need:

$$t_{tot} \leq \frac{f \times n}{r}$$

For a system with 10^6 nodes, with a rate of 200 events/s, and $f = 1\%$, we get a time interval as large as $50s$ to propagate all information. Note also that if r is linearly proportional to n, then t_{tot} is independent of n. It is only a function of the desired success rate.

3.2 Slices and Units

Our system performance depends on the number of slices and units:

1. k is the number of slices the ring is divided into.
2. u is the number of units in a slice.

Parameters k and u determine the expected unit size. This in turn determines t_{small}, the time it takes for information to propagate from a unit leader to all members of a unit, given an assumption about h, the frequency of keep-alive probes. From t_{small} we can determine t_{big} from our calculated value for t_{tot}, given choices of values for t_{wait} and t_{detect}. (Recall that $t_{tot} = t_{detect} + t_{big} + t_{wait} + t_{small}$.)

To simplify the analysis we will choose values for h, t_{detect}, and t_{wait}. As a result our analysis will be concerned with just two independent variables, k and u, given a particular choice of values for n, r, and f. We will use one second for both h and t_{wait}. This is a reasonable decision since the amount of data being sent in probes and messages to unit leaders is large enough to make the overhead in these messages small (e.g., information about 20 events will be sent in a system with 10^5 nodes). Note that with this choice of h, t_{small} will be half the unit size. We will use three seconds for t_{detect} to account for the delay in detecting a missed keep-alive message and a few probes to confirm the event.

3.3 Cost Analysis

Our goal is to choose values for k and u in a way that reduces bandwidth utilization. In particular we are concerned with minimizing bandwidth use at the slice leaders, since they have the most work to do in our approach.

Bandwidth is consumed both to propagate the actual data, and because of the message overhead. m bytes will be required to describe an event, and the overhead per message will be v.

There are four types of communication in our system.

1. *Keep-alive messages:* Keep-alive messages form the base level communication between a node and its predecessor and successor. These messages include information about recent events. As described in Section 2, our system avoids sending redundant information in these messages by controlling the direction of information flow (from unit leader to unit members) and by not sending information across unit boundaries.

   Since keep-alive messages are sent every second, every node that is not on the edge of a unit will send and acknowledge an aggregate message containing, on average, r events. The size of this message is therefore $r \cdot m + v$ and the size of the acknowledgement is v.

2. *Event notification to slice leaders:* Whenever a node detects an event, it sends a notification to its slice leader. The expected number of events per second in a slice is $\frac{r}{k}$. The downstream bandwidth utilization on slice leaders is therefore $\frac{r \cdot (m+v)}{k}$. Since each message must be acknowledged, the upstream utilization is $\frac{r \cdot v}{k}$.

3. *Messages exchanged between slice leaders:* Each message sent from one slice leader to another batches together events that occurred in the last t_{big} seconds in the slice. The typical message size is, therefore, $\frac{r}{k} \cdot t_{big} \cdot m + v$ bytes. During any t_{big} period, a slice leader sends this message to all other slice leaders (k 1 of them), and receives an acknowledgement from each of them. Since each slice leader receives as much as it gets on average, the upstream and downstream use of bandwidth is symmetric. Therefore, the bandwidth utilization (both upstream and downstream) is

   $$\left(\frac{r \cdot m}{k} + \frac{2 \cdot v}{t_{big}}\right) \cdot (k \quad 1)$$

4. *Messages from slice leaders to unit leaders:* Messages received by a slice leader are batched for one second and then forwarded to unit leaders. In one second, r events happen and therefore the aggregate message size is $(r \cdot m + v)$ and the bandwidth utilization is

   $$(r \cdot m + v) \cdot u$$

Table 1 summarizes the net bandwidth use on each node. To clarify the presentation, we have removed insignificant terms from the expressions.

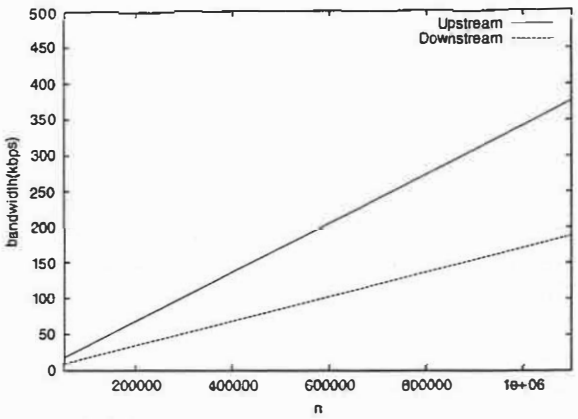


Figure 3: Bandwidth use on a slice leader with $r \propto n$

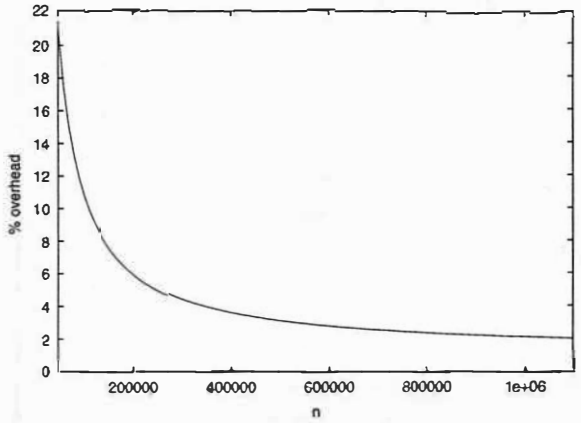


Figure 4: Aggregate bandwidth overhead of the scheme as a percentage of the theoretical optimum

	Upstream	Downstream
Slice Leader	$r \cdot m \cdot (u+2) + \frac{2 \cdot v \cdot k}{t_{big}}$	$r \cdot m + \frac{2 \cdot v \cdot k}{t_{big}}$
Unit Leader	$2 \cdot r \cdot m + 3 \cdot v$	$r \cdot m + 2 \cdot v$
Other nodes	$r \cdot m + 2 \cdot v$	$r \cdot m + 2 \cdot v$

Table 1: Summary of bandwidth use

Using these formulas we can compute the load on non-slice leaders in a particular configuration. In this computations we use $m = 20$ bytes and $v = 40$ bytes. In a system with 10^5 nodes, we see that the load on an ordinary node is 3.84 kbps and the load on a unit leader is 7.36 kbps upstream and 3.84 kbps downstream. For a system with 10^6 nodes, these numbers become 38.4 kbps, 73.6 kbps, and 38.4 kbps respectively.

From the table it is clear that the upstream bandwidth required for a slice leader is likely to be the dominating and limiting term. Therefore, we shall choose parameters that minimize this bandwidth. By simplifying the expression and using the interrelationship between u and t_{big} (explained in Section 3.2) we get a function that depends on two independent variables k and u. By analyzing the function, we deduce that the minimum is achieved for the following values:

$$k = \sqrt{\frac{r \cdot m \cdot n}{4 \cdot v}}$$

$$u = \sqrt{\frac{4 \cdot v \cdot n}{r \cdot m \cdot (t_{tot} \quad t_{wait} \quad t_{detect})^2}}$$

These formulas allow us to compute values for k and u. For example in a system of 10^5 nodes we want roughly 500 slices each containing 5 units. In a system of 10^6 nodes, we still have 5 units per slice, but now there are 5000 slices.

Given values for k and u we can compute the unit size and this in turn allows us to compute t_{small} and t_{big}. We find that we use least bandwidth when

$$t_{small} = t_{big}$$

Thus, we choose 23 seconds for t_{big} and 23 seconds for t_{small}.

Given these values and the formulas given in Table 1, we can plot the bandwidth usage per slice leader in systems of various sizes. The results of this calculation are shown in Figure 3. Note that the load increases only linearly with the size of the system. The load is quite modest in a system with 10^5 nodes (35 kbps upstream bandwidth), and therefore even nodes behind cable modems can act as slice leaders in such a system. In a system with 10^6 nodes the upstream bandwidth required at a slice leader is approximately 350 kbps. Here it would be more appropriate to limit slice leaders to being machines on reasonably provisioned local area networks. For larger networks, the bandwidth increases to a point where a slice leader would need to be a well-provisioned node.

Figure 4 shows the percentage overhead of this scheme in terms of aggregate bandwidth used in the system with respect to the hypothetical optimum scheme with zero overhead. In such a scheme scheme, the cost is just the total bandwidth used in sending r events to every node in the system every second. i.e., $r \cdot n \cdot m$. Note that the overhead in our system comes from the per-message protocol overhead. The scheme itself does not propagate any redundant information. We note that the overhead is approximately 20% for a 10^5 sized system and goes down to 2% for 10^6 sized system. This result is reasonable because messages get larger and the overhead becomes less significant as system size increases.

4 Related Work

Rodrigues et al. [7] proposed a single hop distributed hash table but they assumed a much smaller peer dynamics, like that in a corporate environment, and therefore did not have to deal with the difficulties of rapidly handling a large number of membership changes with efficient bandwidth usage. Douceur et al. [2] present a system that routes in a constant number of hops, but that design assumes smaller peer dynamics and searches can be lossy.

Kelips [3] uses $\sqrt{n}$ sized tables per node and a gossip mechanism to propagate event notifications to provide constant time lookups. Their lookups, however, are constant time only when the routing table entries are reasonably accurate. As seen before, these systems are highly dynamic and the accuracy of the tables depends on how long it takes for the system to converge after an event. The expected convergence time for an event in Kelips is $O(\sqrt{n} \times \log^3(n))$. While this will be tens of seconds for small systems of around a 1000 nodes, for systems having 10^5 to 10^6 nodes, it takes over an hour for an event to be propagated through the system. At this rate, a large fraction of the routing entries in each table are likely to be stale, and a correspondingly large fraction of queries would fail on their first attempt.

Mahajan et al. [5] also derive analytic models for the cost of maintaining reliability in the Pastry [8] peer-to-peer routing algorithm in a dynamic setting. This work differs substantially from ours in that the nature of the routing algorithms is quite different – Pastry uses only $O(log\ N)$ state but requires $O(log\ N)$ hops per lookup – and they focus their work on techniques to reduce their (already low) maintenance cost.

Liben-Nowell et al. [4] provide a lower-bound on the cost of maintaining routing information in peer-to-peer networks that try to maintain topological structure. We are designing a system that requires significantly larger bandwidth than in the lower bound because we aim to achieve a much lower lookup latency.

5 Conclusion

This paper shows that maintaining only a small amount of routing state at each node is not necessary in a dynamic peer-to-peer system. We present a design for a system that maintains complete membership information with reasonable bandwidth requirements.

Currently deployed and proposed systems vary greatly in size and membership behavior. Corporate and academic environments have far fewer configuration events; e.g., half of 64,610 machines probed in a software company are up over 95% of the time [1]. If we design our system to deal with these relatively stable environments, we will have much lower bandwidth requirements.

For systems of size much greater than a million nodes, routing tables become large and it may not be desirable to keep them completely in primary memory. In such a deployment scenario, we may want to use a two-hop routing scheme instead: The querying node contacts a node in the slice containing the target node. That node then redirects the query to the target node. In such a scheme, the querying node needs to be aware of only a few nodes of other slices, leading to smaller routing tables. It is not difficult to adapt our approach to such a scheme, with large savings in bandwidth because very little inter-slice information needs to be propagated.

Currently peer-to-peer storage systems have high lookup latency and are therefore only well-suited for applications that do not mind high-latency store and retrieve operations (e.g., backups) or that store and retrieve massive amounts of data (e.g., a source tree distribution). Moving to more efficient routing removes this constraint. This way we can enable a much larger class of applications for peer-to-peer systems.

Acknowledgements

This research is supported by DARPA under contract F30602-98-1-0237 and NSF Grant IIS-9802066. R. Rodrigues was supported by a Praxis XXI fellowship.

References

[1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *ACM SIGMETRICS*, 2000.

[2] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, July 2002.

[3] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *IPTPS*, 2003.

[4] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, 2002.

[5] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS*, 2003.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, Aug. 2001.

[7] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *SIGOPS European Workshop*, 2002.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, Nov. 2001.

[9] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, Jan. 2002.

[10] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*, Aug. 2001.

[11] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

An Analysis of Compare-by-hash

Val Henson
Sun Microsystems
vhenson@eng.sun.com

Abstract

Recent research has produced a new and perhaps dangerous technique for uniquely identifying blocks that I will call *compare-by-hash*. Using this technique, we decide whether two blocks are identical to each other by comparing their hash values, using a collision-resistant hash such as SHA-1[5]. If the hash values match, we assume the blocks are identical without further ado. Users of compare-by-hash argue that this assumption is warranted because the chance of a hash collision between any two randomly generated blocks is estimated to be many orders of magnitude smaller than the chance of many kinds of hardware errors. Further analysis shows that this approach is not as risk-free as it seems at first glance.

1 Introduction

Compare-by-hash is a technique that trades on the insight that applications frequently read or write data that is identical to already existing data. Rather than read or write the data a second time to the disk, network, or memory, we should use the instance of the data that we already have. Using a collision-resistant hash, we can quickly determine with a high degree of accuracy whether two blocks are identical by comparing only their hashes and not their contents. After making a few assumptions, we can estimate that the chance of a hash collision is much lower than the chance of a hardware error, and so many feel comfortable neglecting the possibility of a hash collision.

Compare-by-hash is accepted by some computer scientists and has been implemented in several different projects: rsync[14], a utility for synchronizing files, LBFS[4], a distributed file system, Stanford's virtual computer migration project[9], Venti[6], a block archival system, Pastiche[3], an automated backup system, and OpenCM[11], a configuration management system. However, I believe some publications overstate the acceptance of compare-by-hash, claiming that it is "customary"[3] or a "widely-accepted practice"[4] to assume hashes never collide in this context. An informal survey of my colleagues reveals that many computer scientists are still either unaware of compare-by-hash or disagree with the technique strongly. Since adoption of compare-by-hash has the potential to change the face of operating systems design and implementation, it should be the subject of more criticism and peer review before being accepted as a general purpose computing technique for critical applications.

In this position paper, I hope to begin an in-depth discussion of compare-by-hash. Section 2 reviews the traditional uses of hashing, followed by a more detailed description of compare-by-hash in Section 3. Section 4 will raise some questions about the use of compare-by-hash as a general-purpose technique. Section 5 will propose some alternatives to compare-by-hash, and Section 6 will summarize my findings and make recommendations.

2 Traditional applications of hashing

The review in this section may seem tedious and unnecessary, but I believe that a clear understanding of how hashing has been used in the past is necessary to understand how compare-by-hash differs.

A **hash function** maps a variable length input string to fixed length output string — its **hash value**, or **hash** for short. If the input is longer than the output, then some inputs must map to the same output — a **hash collision**. Comparing the hash values for two inputs can give us one of two answers: the inputs are definitely not the same, or there is a possibility that they are the same. Hashing as we know it is used for performance improvement, error checking, authentication, and encryption. One example of a performance improvement is the common hash table, which uses a hash function to index into the correct bucket in the hash table, followed by comparing each element in the bucket to find a match. In error checking, hashes (checksums, message digests, etc.) are used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on

downloaded files[1]. In this case, the hash provides additional assurance that the data we received is correct. Finally, hashes are used to authenticate messages. In this case, we are trying to protect the original input from tampering, and we select a hash that is strong enough to make malicious attack infeasible or unprofitable.

3 Compare-by-hash in detail

Compare-by-hash is a technique used when the payoff of discovering identical blocks is worth the computational cost of computing the hash of a block. In compare-by-hash, we assume hash collisions never occur, so we can treat the hash of a block as a unique id and compare only the hashes of blocks rather than the contents of blocks. For example, we can use compare-by-hash to reduce bandwidth usage. Before sending a block, the sender first transmits the hash of the block to the receiver. The receiver checks to see if it has a local block with the same hash value. If it does, it assumes that it is the same block as the sender's, without actually comparing the two input blocks. In the case of a 4096 byte block and a 160 bit hash value, this system can reduce network traffic from 4096 bytes to 20 bytes, or about a 99.5% savings in bandwidth.

This is an incredible savings! The cost, of course, is the risk of a hash collision. We can reduce that risk by choosing a collision-resistant hash. From a cryptographic point of view, collision resistance means that it is difficult to find two inputs that hash to the same output. By implication, the range of hash values must be large enough that a brute-force attack to find collisions is "difficult."[2] Cryptologists have given us several algorithms that appear to have this property, although so far, only SHA-1 and RIPEMD-160 have stood up to careful analysis[8].

With a few assumptions, we can arrive at an estimate for the risk of a hash collision. We assume that the inputs to the hash function are random and uniformly distributed, and the output of the hash function is also random and uniformly distributed. Let n be the number of input blocks, and let b be the number of bits in the hash output. As a function of the number of input blocks, n, the probability that we will encounter one or more collisions is $1-(1-2^{-b})^n$. This is a difficult number to calculate when b is 160, but we can use the "birthday paradox"[3] to calculate how many inputs will give us a 50% chance of finding a collision. For a 160-bit output, we will need about $2^{160/2}$ or 2^{80} inputs to have a 50% chance of a collision. Put another way, we expect with about 48 nines ($1-2^{-160}$) of certainty that any two randomly chosen inputs will not collide, whereas empirical measurements tell us we only have perhaps 8 or 9 nines of certainty that we will not encounter an undetected TCP error when we transmit the block[13]. In the face of much larger sources of potential error, the error added by compare-by-hash appears to be negligible.

Now that we've described compare-by-hash in more detail, it should be clear how compare-by-hash and traditional hashing differ: No known previous uses of hashing skip the step of directly comparing the inputs for performance reasons. The only case in which we do skip that step is authentication, because we can't compare the inputs directly due to the lack of a secure channel. Compare-by-hash sets a new precedent and so does not yet enjoy the acceptance of established uses of hashing.

4 Questions about compare-by-hash

What appears to be a fall of manna from heaven should be examined a little more closely before compare-by-hash is accepted into the computer scientist's tricks of the trade. In the following section, I will re-examine the assumptions we made earlier when justifying the use of compare-by-hash.

4.1 Randomness of input

In Section 3, we calculated the probability of a hash collision under the assumption that our inputs were random and uniformly distributed. While this assumption simplifies the math, it is also **wrong**.

Real data is not random, unless all applications produce random data. This may seem like a trivial and facile statement, but it is actually the key insight into the weakness of compare-by-hash. If real data were actually random, each possible input block would be equally likely to occur, whereas in real data, input blocks that contain only ASCII characters or begin with an ELF header are more common than in random data. Knowing that real data isn't

[1] MD5 checksums are designed to detect intentional tampering as well.

[2] A **cryptographically secure hash** is defined as a hash with no known method better than brute force for finding collisions.

[3] The "birthday paradox" is best illustrated by the question, "How many people do you need in a room to have a 50% or greater chance that two of them have the same birthday?" The answer is 23 (assuming that birthdays are uniformly distributed and neglecting leap years). This is easier to understand if you realize that there are $23 \times (22/2) = 253$ different *pairs* of people in the room.

random, can we think of some cases where it is non-random in an interesting way?

Consider an application, let's call it SHA1@home, that attempts to find a collision in the SHA-1 hash function. SHA1@home is a distributed application, so it runs many instances in parallel on many machines, using a distributed file system to share data when necessary. When two inputs are found that hash to the same value, one program reads and compares both input blocks to find out if they differ. If the file system uses compare-by-hash with SHA-1 and the same block size as the inputs for SHA1@home, this application will be unable to detect a collision, ever. For example, if SHA1@home used a 2KB block size, it would run incorrectly if it used LBFS as the underlying file system[4].

This is only one very crude, very simple example of an entire class of applications that are very useful, especially to cryptanalysts. In their 1998 paper, Chabaud and Joux implemented several programs designed specifically to find collisions in various hashing algorithms, including SHA-0 and several relatives. They end by hinting at avenues of research for attacking SHA-1[2]. Somewhat ironically, this paper is referenced by one of the papers using compare-by-hash[9].

4.2 Cryptographic hashes — one size fits all?

Collision-resistant hashes were originally developed for use in cryptosystems. Is a hash intended for cryptography also good for use in systems with different characteristics?

Cryptographic hashes are short-lived. Data is forever, secrecy is not. The literature is rife with examples of cryptosystems that turned out to not be nearly as secure as we thought. Weakness are frequently discovered within a few years of a cryptographic hash's introduction[2, 8, 10]. On the other hand, lifetimes of operating systems, file systems, and file transfer protocols are frequently measured in decades. Solaris, FFS, and ftp come to mind immediately. Cryptologists choose algorithms based on how long they want to keep their data secure, while computer scientists should choose their algorithms based on how long they want to keep their data, period. (Cryptologists may desire to keep data secure for decades, but most would not expect their current algorithms to actually accomplish this goal.)

[4] LBFS uses variable sized blocks, but has minimum block size of 2KB to avoid pathologically small block sizes[4]

Obsolecence can occur overnight. A related consideration is how quickly obsolescence occurs for cryptosystems. In operating systems, we are used to systems slowing and gracefully obsolescing over a period of years. Cryptosystems can go from state-of-the-art to completely useless overnight.

Obsolescence is inevitable. Large governments, corporations, and scientists all have a huge incentive to analyze and break cryptographic hashes. We have no proof that any particular hash, much less SHA-1, is "unbreakable." At the same time, history tells us that we should expect any popular cryptographic hash to be broken within a few years of its introduction. If anyone had built a distributed file system using compare-by-hash and MD4, it would already be unusable today, due to known attacks that take seconds to find a collision using a personal computer. MD5 appears to be well on its way to unusability as well[8].

Upgrade strategy required. Given that our hash algorithms will be obsolete within a few years, systems using compare-by-hash need to have a concrete upgrade plan for what happens when anyone with a PC can generate a hash collision. Upgrade will be more difficult if any hash collisions have occurred, because part of your data will now be corrupted, possibly a very important part of your data.

4.3 Silent, deterministic, hard-to-fix errors

Ordinarily, anyone who discovered two inputs that hash to the same SHA-1 output would become world-famous overnight. On a system using compare-by-hash, that person would instead just silently read or overwrite the wrong data (which is more than a bug, it's a security hole). To understand why silent errors are so bad, think about silent disk corruption. Sometimes the corruption goes undetected until long after the last backup with correct data has been destroyed.

In addition, any two inputs that hash to the same value will always be treated incorrectly, whereas most hardware errors are transitory and data-independent. Redundant disks or servers provide no protection against data-dependent, deterministic errors. To avoid this, we could add a random seed every time we compute the hash, but we won't save anything except in the most extreme cases if we have to recompute hashes on every candidate local block every time we compare a block.

Once a hash collision has been found and a demonstrably buggy test program created using the collid-

ing inputs, how will you fix the bug? Usually, the response to a test program that demonstrates a bug in the system is to fix the bug. In this case, the underlying algorithm is the bug.

4.4 Comparing probabilities

One of the primary arguments for compare-by-hash is a simple comparison of the probability of a hash collision (very low) and the probability of some common hardware error (also low but much higher). To show that we cannot directly compare the probability of a deterministic, data-dependent error with the probability of nondeterministic, data-independent error, let's construct a hash function that has the same collision probability as SHA-1 but, when used in compare-by-hash, will be a far more common source of error than any normal hardware error.

Define VAL-1(x) as follows:

$$\text{VAL-1}(x) = \begin{cases} x > 0 & : \text{SHA-1}(x) \\ x = 0 & : \text{SHA-1}(1) \end{cases}$$

In other words, VAL-1 is SHA-1 except that the first two inputs map to the same output. This function has an almost identical probability of collision as SHA-1, but it is completely unsuitable for use in compare-by-hash. The point of this example is not that bad hash functions will result in errors, but that we can't directly compare the probability of a hash collision with the probability of a hardware error. If we could, VAL-1 and SHA-1 would be equally good candidates for compare-by-hash. The relationship between the probability of a hash collision and the probability of a hardware error must be more complicated than a straightforward comparison can reveal.

4.5 Software and reliability

On a more philosophical note, should software improve on hardware reliability or should programmers accept hardware reliability as an upper bound on total system reliability? What would we think of a file system that had a race condition that was triggered less often than disk I/O errors? What if it lost files only slightly less often than users accidentally deleted them? Once we start playing the game of error relativity, where do we stop? Current software practices suggest that most programmers believe software should improve reliability — hence we have TCP checksums, asserts for impossible hardware conditions, and handling of I/O errors. For example, the empirically observed rate of undetected errors in TCP packets is about 0.0000005%[13]. We could dramatically improve that rate by sending both the block and its SHA-1 hash, or we could slightly worsen that rate by sending only the hash.

4.6 When is compare-by-hash appropriate?

Taking all this into account, when is it reasonable to use compare-by-hash? For one, users of software should know when they are getting best effort and when they are getting correctness. When using rsync, the user knows that there is a tiny but real possibility of an incorrect target file (in rsync's case, the user has only to read the man page). When using a file system, or incurring a page fault, users expect to get exactly the data they wrote, all the time. Another consideration is whether other users share the "address space" produced by compare-by-hash. If only trusted users write data to the system, they don't have to worry about maliciously generated collisions and can avoid known collisions. By these standards, rsync is an appropriate use of compare-by-hash, whereas LBFS, Venti, Pastiche, and Stanford's virtual machine migration are not.

5 Alternatives to compare-by-hash

The alternatives to compare-by-hash can be summarized as "Keep some state!" Compare-by-hash attempts to establish similarities between two unknown sets of blocks. If we keep track of which blocks we are sure are identical (because we directly compared them), we don't have to guess. Unfortunately, keeping state is hard. Part of the popularity of compare-by-hash is undoubtably due to its ease of implementation compared to a stateful solution. However, simplicity of implementation should not come at the cost of correctness.

One of the applications of compare-by-hash is reducing network bandwidth used by distributed file systems. To accomplish nearly the same effect, we can resolve to only send any particular block over the link once, keeping sent and received data in a cache in both sender and receiver. Before sending a block, the sender checks to see if it has already sent the block and if so, sends the block id rather than the block itself. This idea is proposed by Spring and Wetherall in [12]. We might also agree in advance on certain universal block ids, for example, block id 0 is always the zero block of length 4096 bytes. The initial start-up cost is higher, depending on the degree of actually shared blocks between the two machines, but after cache warm-up, performance should be quite similar to compare-by-hash.

In combination with an intelligent blocking tech-

nique, such as Rabin fingerprints[7], which divide up blocks at "anchor" points (patterns in the input) rather than at fixed intervals, we can experiment with byte and block level differencing techniques that require similar amounts of computation time as computing cryptographic hashes. Using fingerprints to determine block boundaries allows us to more easily detect insertions and deletions within blocks.

Compression may still have more mileage left in it, since we are willing to trade off large amounts of computation for reduced bandwidth. We might try compressing with several different algorithms optimized for different inputs.

5.1 Existence proof: Rsync vs. BitKeeper

As an example of a system that improves on compare-by-hash while retaining correctness, compare rsync and BitKeeper[1], a commercial source configuration management tool. They both solve the problem of keeping several source code trees in sync. (We will ignore the unrelated features of BitKeeper, such as versioning, in this comparison.) Rsync is stateless; it has no a priori knowledge of the relationship between two source code trees. It uses compare-by-hash to determine which blocks are different between the two trees and sends only the blocks with different hashes. BitKeeper keeps state about each file under source control and knows what changes have been made since the last time each tree was synchronized. When synchronizing, it sends only the differences since the last synchronization occurred, in compressed form. In comparison to rsync, BitKeeper provides similar and sometimes better bandwidth usage when simply synchronizing two trees without resorting to compare-by-hash. Improvements BitKeeper provides over rsync include elimination of reverse updates (synchronizing in the wrong direction and losing your changes), automerging algorithms optimized for source code (so trees can be updated in parallel and then synchronized), and intelligent handling of metadata operations such as renaming of files (which rsync sees as deletion and creation of files).

With a little more programming effort, we can get the bandwidth reduction promised by compare-by-hash without sacrificing correctness and at the same time adding functionality. Compare-by-hash still has applications in areas where statelessness and low bandwidth are more important than correctness of data referenced, and users are aware of the risk they are taking, as in rsync.

6 Conclusion

Use of compare-by-hash is justified by mathematical calculations based on assumptions that range from unproven to demonstrably wrong. The short lifetime and fast transition into obsolescence of cryptographic hashes makes them unsuitable for use in long-lived systems. When hash collisions do occur, they cause silent errors and bugs that are difficult to repair. What should worry computer scientists the most about compare-by-hash is that real people are running real workloads that will execute incorrectly on systems using compare-by-hash. Perhaps research would be better directed towards alternatives to or improvements on compare-by-hash that avoid the problems described. At the very least, future research using compare-by-hash should include a more careful analysis of the risk of hash collisions.

7 Acknowledgments

Many people joined in on (both sides of) the discussion that led to this paper and provided helpful comments on drafts, including Jonathan Adams, Matt Ahrens, Jeff Bonwick, Bryan Cantrill, Miguel Castro, Whit Diffie, Marius Eriksen, Barry Hayes, Richard Henderson, Larry McVoy, Dave Powell, Bart Smaalders, Niraj Tolia, Vernor Vinge, and Cynthia Wong.

References

[1] Bitmover, Inc. Bitkeeper - the scalable distributed software configuration management system. http://www.bitkeeper.com.

[2] Florent Chabuad and Antoine Joux. Differential collisions in SHA-0. In *Proceedings of CRYPTO '98, 18th Annual International Cryptology Conference*, pages 56–71, 1998.

[3] Landon P. Cox, Christoper D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[4] Athicha Muthitacharoen, Benjie Chen, and David Maziéres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[5] National Institute of Standards and Technology. *FIPS Publication 180–1: Secure Hash Standard*, 1995.

[6] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 2002.

[7] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR–15–81, Center for Research in Computer Technology, Harvard University, 1981.

[8] B. Van Rompay, B. Preneel, and J. Vandewalle. On the security of dedicated hash functions. In *19th Symposium on Information Theory in the Benelux*, 1998.

[9] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[10] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., second edition, 1996.

[11] Jonathan S. Shapiro and John Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *Proceedings of the 2002 USENIX Technical Conference, FREENIX Track*, 2002.

[12] Neil T. Spring and David Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proceedings of the 2000 ACM SIGCOMM Conference*, 2000.

[13] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *Proceedings of the 2000 ACM SIGCOMM Conference*, 2000.

[14] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, 1999.

Why Events Are A Bad Idea (for high-concurrency servers)

Rob von Behren, Jeremy Condit and Eric Brewer
Computer Science Division, University of California at Berkeley
{jrvb, jcondit, brewer}@cs.berkeley.edu
http://capriccio.cs.berkeley.edu/

Abstract

Event-based programming has been highly touted in recent years as the best way to write highly concurrent applications. Having worked on several of these systems, we now believe this approach to be a mistake. Specifically, we believe that threads can achieve all of the strengths of events, including support for high concurrency, low overhead, and a simple concurrency model. Moreover, we argue that threads allow a simpler and more natural programming style.

We examine the claimed strengths of events over threads and show that the weaknesses of threads are artifacts of specific threading implementations and not inherent to the threading paradigm. As evidence, we present a user-level thread package that scales to 100,000 threads and achieves excellent performance in a web server. We also refine the duality argument of Lauer and Needham, which implies that good implementations of thread systems and event systems will have similar performance. Finally, we argue that compiler support for thread systems is a fruitful area for future research. It is a mistake to attempt high concurrency without help from the compiler, and we discuss several enhancements that are enabled by relatively simple compiler changes.

1 Introduction

Highly concurrent applications such as Internet servers and transaction processing databases present a number of challenges to application designers. First, handling large numbers of concurrent tasks requires the use of scalable data structures. Second, these systems typically operate near maximum capacity, which creates resource contention and high sensitivity to scheduling decisions; overload must be handled with care to avoid thrashing. Finally, race conditions and subtle corner cases are common, which makes debugging and code maintenance difficult.

Threaded servers have historically failed to meet these challenges, leading many researchers to conclude that event-based programming is the best (or even only) way to achieve high performance in highly concurrent applications. The literature gives four primary arguments for the supremacy of events:

- Inexpensive synchronization due to cooperative multitasking;
- Lower overhead for managing state (no stacks);
- Better scheduling and locality, based on application-level information; and
- More flexible control flow (not just call/return).

We have made extensive use of events in several high-concurrency environments, including Ninja [16], SEDA [17], and Inktomi's Traffic Server. In working with these systems, we realized that the properties above are not restricted to event systems; many have already been implemented with threads, and the rest are possible.

Ultimately, our experience led us to conclude that event-based programming is the wrong choice for highly concurrent systems. We believe that (1) threads provide a more natural abstraction for high-concurrency servers, and that (2) small improvements to compilers and thread runtime systems can eliminate the historical reasons to use events. Additionally, threads are more amenable to compiler-based enhancements; we believe the right paradigm for highly concurrent applications is a thread package with better compiler support.

Section 2 compares events with threads and rebuts the common arguments against threads. Next, Section 3 explains why threads are particularly natural for writing high-concurrency servers. Section 4 explores the value of compiler support for threads. In Section 5, we validate our approach with a simple web server. Finally, Section 6 covers (some) related work, and Section 7 concludes.

2 Threads vs. Events

The debate between threads and events is a very old one. Lauer and Needham attempted to end the discussion in 1978 by showing that message-passing systems and process-based systems are duals, both in terms of program structure and performance characteristics [10]. Nonetheless, in recent years many authors have declared the need for event-driven programming for highly concurrent systems [11, 12, 17].

Events	Threads
event handlers	monitors
events accepted by a handler	functions exported by a module
SendMessage / AwaitReply	procedure call, or fork/join
SendReply	return from procedure
waiting for messages	waiting on condition variables

Figure 1: A selection of dual notions in thread and event systems, paraphrased from Lauer and Needham. We have converted their terminology to contemporary terms from event-driven systems.

2.1 Duality Revisited

To understand the threads and events debate, it is useful to reexamine the duality arguments of Lauer and Needham. Lauer and Needham describe canonical threaded and message-passing (i.e., event-based) systems. Then, they provide a mapping between the concepts of the two regimes (paraphrased in Figure 1) and make the case that with proper implementations, these two approaches should yield equivalent performance. Finally, they argue that the decision comes down to which paradigm is more natural for the target application. In the case of high-concurrency servers, we believe the thread-based approach is preferable.

The message-passing systems described by Lauer and Needham do not correspond precisely to modern event systems in their full generality. First, Lauer and Needham ignore the cooperative scheduling used by events for synchronization. Second, most event systems use shared memory and global data structures, which are described as atypical for Lauer and Needham's message-passing systems. In fact, the only event system that really matches their canonical message-passing system is SEDA [17], whose stages and queues map exactly to processes and message ports.[1]

Finally, the performance equivalence claimed by Lauer and Needham requires equally good implementations; we don't believe there has been a suitable threads implementation for very high concurrency. We demonstrate one in the next section, and we discuss further enhancements in Section 4.

In arguing that performance should be equivalent, Lauer and Needham implicitly use a graph that we call a *blocking graph*. This graph describes the flow of control through an application with respect to blocking or yielding points. Each node in this graph represents a blocking or yielding point, and each edge represents the code that is executed between two such points. The Lauer-Needham duality argument essentially says that duals have the same graph.

The duality argument suggests that criticisms of thread performance and usability in recent years have been motivated by problems with *specific* threading packages, rather than with threads in general. We examine the most common criticisms below.

[1] Arguably, one of SEDA's contributions was to return event-driven systems to the "good practices" of Lauer-Needham.

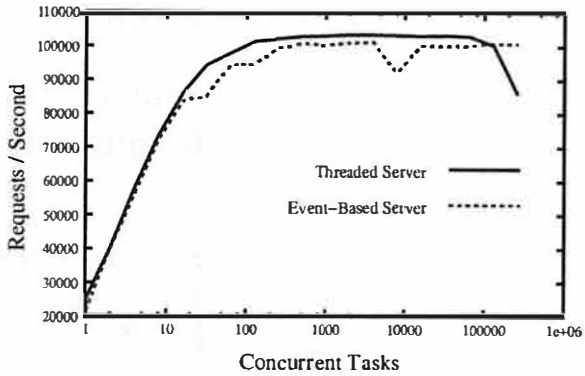


Figure 2: A repeat of the threaded server benchmark from the SEDA paper [17]. The threaded server uses a preallocated thread pool to process requests, while the event server uses a single thread to pull items from the queue. Requests are internally generated to avoid network effects. Each request consists of an 8K read from a cached disk file.

2.2 "Problems" with Threads

Performance. *Criticism: Many attempts to use threads for high concurrency have not performed well.* We don't dispute this criticism; rather, we believe it is an artifact of poor thread implementations, at least with respect to high concurrency. None of the currently available thread packages were designed for both high concurrency and blocking operations, and thus it is not surprising that they perform poorly.

A major source of overhead is the presence of operations that are $O(n)$ in the number of threads. Another common problem with thread packages is their relatively high context switch overhead when compared with events. This overhead is due to both preemption, which requires saving registers and other state during context switches, and additional kernel crossings (in the case of kernel threads).

However, these shortcomings are not intrinsic properties of threads. To illustrate this fact, we repeated the SEDA threaded server benchmark [17] with a modified version of the GNU Pth user-level threading package, which we optimized to remove most of the $O(n)$ operations from the scheduler. The results are shown in Figure 2. Our optimized version of Pth scales quite well up to 100,000 threads, easily matching the performance of the event-based server.

Control Flow. *Criticism: Threads have restrictive control flow.* One argument against threaded programming is that it encourages the programmer to think too linearly about control flow, potentially precluding the use of more efficient control flow patterns. However,

complicated control flow patterns are rare in practice. We examined the code structure of the Flash web server and of several applications in Ninja, SEDA, and TinyOS [8, 12, 16, 17]. In all cases, the control flow patterns used by these applications fell into three simple categories: call/return, parallel calls, and pipelines. All of these patterns can be expressed more naturally with threads.

We believe more complex patterns are not used because they are difficult to use well. The accidental non-linearities that often occur in event systems are already hard to understand, leading to subtle races and other errors. Intentionally complicated control flow is equally error prone.

Indeed, it is no coincidence that common event patterns map cleanly onto the call/return mechanism of threads. Robust systems need acknowledgements for error handling, for storage deallocation, and for cleanup; thus, they need a "return" even in the event model.

The only patterns we considered that are less graceful with threads are dynamic fan-in and fan-out; such patterns might occur with multicast or publish/subscribe applications. In these cases, events are probably more natural. However, none of the high-concurrency servers that we studied used these patterns.

Synchronization. *Criticism: Thread synchronization mechanisms are too heavyweight.* Event systems often claim as an advantage that cooperative multitasking gives them synchronization "for free," since the runtime system does not need to provide mutexes, handle wait queues, and so on [11]. However, Adya *et al.* [1] show that this advantage is really due to cooperative multitasking (i.e., no preemption), not events themselves; thus, cooperative thread systems can reap the same benefits. It is important to note that in either regime, cooperative multitasking only provides "free" synchronization on uniprocessors, whereas many high-concurrency servers run on multiprocessors. We discuss compiler techniques for supporting multiprocessors in Section 4.3.

State Management. *Criticism: Thread stacks are an ineffective way to manage live state.* Threaded systems typically face a tradeoff between risking stack overflow and wasting virtual address space on large stacks. Since event systems typically use few threads and unwind the thread stack after each event handler, they avoid this problem. To solve this problem in threaded servers, we propose a mechanism that will enable dynamic stack growth; we will discuss this solution in Section 4.

Additionally, event systems encourage programmers to minimize live state at blocking points, since they require the programmer to manage this state by hand. In contrast, thread systems provide automatic state management via the call stack, and this mechanism can allow programmers to be wasteful. Section 4 details our solution to this problem.

Scheduling. *Criticism: The virtual processor model provided by threads forces the runtime system to be too generic and prevents it from making optimal scheduling decisions.* Event systems are capable of scheduling event deliveries at application level. Hence, the application can perform shortest remaining completion time scheduling, favor certain request streams, or perform other optimizations. There has also been some evidence that events allow better code locality by running several of the same kind of event in a row [9]. However, Lauer-Needham duality indicates that we can apply the same scheduling tricks to cooperatively scheduled threads.

2.3 Summary

The above arguments show that threads can perform at least as well as events for high concurrency and that there are no substantial qualitative advantages to events. The absence of scalable user-level threads has provided the largest push toward the event style, but we have shown that this deficiency is an artifact of the available implementations rather than a fundamental property of the thread abstraction.

3 The Case for Threads

Up to this point, we have largely argued that threads and events are equivalent in power and that threads can in fact perform well with high concurrency. In this section, we argue that threads are actually a more appropriate abstraction for high-concurrency servers. This conclusion is based on two observations about modern servers. First, the concurrency in modern servers results from concurrent requests that are largely independent. Second, the code that handles each request is usually sequential. We believe that threads provide a better programming abstraction for servers with these two properties.

Control Flow. For these high-concurrency systems, event-based programming tends to obfuscate the control flow of the application. For instance, many event systems "call" a method in another module by sending an event and expect a "return" from that method via a similar event mechanism. In order to understand the application, the programmer must mentally match these call/return pairs, even when they are in different parts of the code. Furthermore, these call/return pairs often require the programmer to manually save and restore live state. This process, referred to as "stack ripping" by Adya *et al.* [1], is a major burden for programmers who wish to use event systems. Finally, this obfuscation of the program's control flow can also lead to subtle race conditions and logic errors due to unexpected message arrivals.

Thread systems allow programmers to express control flow and encapsulate state in a more natural manner. Syntactically, thread systems group calls with returns,

making it much easier to understand cause/effect relationships, and ensuring a one-to-one relationship. Similarly, the run-time call stack encapsulates all live state for a task, making existing debugging tools quite effective.

Exception Handling and State Lifetime. Cleaning up task state after exceptions and after normal termination is simpler in a threaded system, since the thread stack naturally tracks the live state for that task. In event systems, task state is typically heap allocated. Freeing this state at the correct time can be extremely difficult because branches in the application's control flow (especially in the case of error conditions) can cause deallocation steps to be missed.

Many event systems, such as Ninja and SEDA, use garbage collection to solve this problem. However, previous work has found that Java's general-purpose garbage collection mechanism is inappropriate for high-performance systems [14]. Inktomi's Traffic Server used reference counting to manage state, but maintaining correct counts was difficult, particularly for error handling.[2]

Existing Systems. The preference for threads is subtly visible even in existing event-driven systems. For example, our own Ninja system [16] ended up using threads for the most complex parts, such as recovery, simply because it was nearly impossible to get correct behavior using events (which we tried first). In addition, applications that didn't need high concurrency were always written with threads, just because it was simpler. Similarly, the FTP server in Harvest uses threads [4].

Just Fix Events? One could argue that instead of switching to thread systems, we should build tools or languages that address the problems with event systems (i.e., reply matching, live state management, and shared state management). However, such tools would effectively duplicate the syntax and run-time behavior of threads. As a case in point, the cooperative task management technique described by Adya *et al.* [1] allows users of an event system to write thread-like code that gets transformed into continuations around blocking calls. In many cases, fixing the problems with events is tantamount to switching to threads.

4 Compiler Support for Threads

Tighter integration between compilers and runtime systems is an extremely powerful concept for systems design. Threaded systems can achieve improved safety and performance with only minor modifications to existing compilers and runtime systems. We describe how this synergy can be used both to overcome limitations in current threads packages and to improve safety, programmer productivity, and performance.

[2]Nearly every release battled with slow memory leaks due to this kind of reference counting; such leaks are often the limiting factor for the MTBF of the server.

4.1 Dynamic Stack Growth

We are developing a mechanism that allows the size of the stack to be adjusted at run time. This approach avoids the tradeoff between potential overflow and wasted space associated with fixed-size stacks. Using a compiler analysis, we can provide an upper bound on the amount of stack space needed when calling each function; furthermore, we can determine which call sites may require stack growth. Recursive functions and function pointers produce additional challenges, but these problems can be addressed with further analyses.

4.2 Live State Management

Compilers could easily purge unnecessary state from the stack before making function calls. For example, temporary variables could be popped before subroutines are called, and the entire frame could be popped in the case of a tail call. Variables with overlapping lifetimes could be automatically reordered or moved off the stack in order to prevent live variables from unnecessarily pinning dead ones in memory. The compiler could also warn the programmer of cases where large amounts of state might be held across a blocking call, allowing the programmer to modify the algorithms if space is an issue.

4.3 Synchronization

Compile-time analysis can reduce the occurrence of bugs by warning the programmer about data races. Although static detection of race conditions is challenging, there has been recent progress due to compiler improvements and tractable whole-program analyses. In nesC [7], a language for networked sensors based on the TinyOS architecture [8], there is support for atomic sections, and the compiler understands the concurrency model. TinyOS uses a mixture of events and run-to-completion threads, and the compiler uses a variation of a call graph that is similar to the blocking graph. The compiler ensures that atomic sections reside within one edge on that graph; in particular, calls within an atomic section cannot yield or block (even indirectly). Compiler analysis can also help determine which atomic sections are safe to run concurrently. This information can then be given to the runtime system to allow safe execution on multiprocessors, thus automating the hand-coded graph coloring technique used in libasync [5].

5 Evaluation

To evaluate the ability of threads to support high concurrency, we designed and implemented a simple (5000 line) user-level cooperative threading package for Linux. Our thread package uses the `coro` coroutine library [15] for minimalist context switching, and it translates blocking I/O requests to asynchronous requests internally. For asynchronous socket I/O, we use the

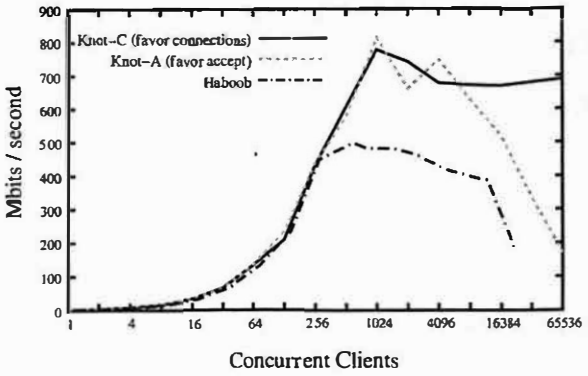


Figure 3: Web server bandwidth versus the number of simultaneous clients. We were unable to run the benchmark for Haboob with more than 16384 clients, as Haboob ran out of memory.

UNIX `poll()` system call, whereas asynchronous disk I/O is provided by a thread pool that performs blocking I/O operations. The library also overrides blocking system calls and provides a simple emulation of pthreads, which allows applications written for our library to compile unmodified with standard pthreads.

With this thread package we wrote a 700-line test web server, Knot. Knot accepts static data requests, allows persistent connections, and includes a basic page cache. The code is written in a clear, straightforward threaded style and required very little performance tuning.

We compared the performance of Knot to that of SEDA's event-driven web server, Haboob, using the test suite used to evaluate SEDA [17]. The `/dev/poll` patch used for the original Haboob tests has been deprecated, so our tests of Haboob used standard UNIX `poll()` (as does Knot). The test machine was a 2x2000 MHz Xeon SMP with 1 GB of RAM running Linux 2.4.20. The test uses a small workload, so there is little disk activity. We ran Haboob with the 1.4 JVM from IBM, with the JIT enabled. Figure 3 presents the results.

We tested two different scheduling policies for Knot, one that favors processing of active connections over accepting new ones (Knot-C in the figure) and one that does the reverse (Knot-A). The first policy provides a natural throttling mechanism by limiting the number of new connections when the server is saturated with requests. The second policy was designed to create higher internal concurrency, and it more closely matches the policy used by Haboob.

Figure 3 shows that Knot and Haboob have the same general performance pattern. Initially, there is a linear increase in bandwidth as the number of simultaneous connections increases; when the server is saturated, the bandwidth levels out. The performance degradation for both Knot-A and Haboob is due to the poor scalability of `poll()`. Using the newer `sys_epoll` system call with Knot avoids this problem and achieves excellent scalability. However, we have used the `poll()` result for comparison, since `sys_epoll` is incompatible with Haboob's socket library. This result shows that a well-designed thread package can achieve the same scaling behavior as a well-designed event system.

The steady-state bandwidth achieved by Knot-C is nearly 700 Mbit/s. At this rate, the server is apparently limited by interrupt processing overhead in the kernel. We believe the performance spike around 1024 clients is due to lower interrupt overhead when fewer connections to the server are being created.

Haboob's maximum bandwidth of 500 Mbit/s is significantly lower than Knot's, because Haboob becomes CPU limited at 512 clients. There are several possible reasons for this result. First, Haboob's thread-pool-per-handler model requires context switches whenever events pass from one handler to another. This requirement causes Haboob to context switch 30,000 times per second when fully loaded—more than 6 times as frequently as Knot. Second, the proliferation of small modules in Haboob and SEDA (a natural outgrowth of the event programming model) creates a large number of module crossings and queuing operations. Third, Haboob creates many temporary objects and relies heavily on garbage collection. These challenges seem deeply tied to the event model; the simpler threaded style of Knot avoids these problems and allows for more efficient execution. Finally, event systems require various forms of run-time dispatch, since the next event handler to execute is not known statically. This problem is related to the problem of ambiguous control flow, which affects performance by reducing opportunities for compiler optimizations and by increasing CPU pipeline stalls.

6 Related Work

Ousterhout [11] made the most well-known case in favor of events, but his arguments do not conflict with ours. He argues that programming with concurrency is fundamentally difficult, and he concludes that cooperatively scheduled events are preferable (for most purposes) because they allow programmers to avoid concurrent code in most cases. He explicitly supports the use of threads for true concurrency, which is the case in our target space. We also agree that cooperative scheduling helps to simplify concurrency, but we argue that this tool is better used in the context of the simpler programming model of threads.

Adya *et al.* [1] cover a subset of these issues better than we have. They identify the value of cooperative scheduling for threads, and they define the term "stack ripping" for management of live state. Our work expands

on these ideas by exploring thread performance issues and compiler support techniques.

SEDA is a hybrid approach between events and threads, using events between stages and threads within them [17]. This approach is quite similar to the message-passing model discussed by Lauer and Needham [10], though Lauer and Needham advocate a single thread per stage in order to avoid synchronization within a stage. SEDA showed the value of keeping the server in its operating range, which it did by using explicit queues; we agree that the various queues for threads *should* be visible, as they enable better debugging and scheduling. In addition, the stage boundaries of SEDA provide a form of modularity that simplifies composition in the case of pipelines. When call/return patterns are used, such boundaries require stack ripping and are better implemented with threads using blocking calls.

Many of the techniques we advocate for improving threads were introduced in previous work. Filaments [6] and NT's Fibers are good examples of cooperative user-level threads packages, although neither is targeted at large numbers of blocking threads. Languages such as Erlang [2] and Concurrent ML [13] include direct support for concurrency and lightweight threading. Bruggeman *et al.* [3] employ dynamically linked stacks to implement one-shot continuations, which can in turn be used to build user-level thread packages. Our contribution is to bring these techniques together in a single package and to make them accessible to a broader community of programmers.

7 Conclusions

Although event systems have been used to obtain good performance in high concurrency systems, we have shown that similar or even higher performance can be achieved with threads. Moreover, the simpler programming model and wealth of compiler analyses that threaded systems afford gives threads an important advantage over events when writing highly concurrent servers. In the future, we advocate tight integration between the compiler and the thread system, which will result in a programming model that offers a clean and simple interface to the programmer while achieving superior performance.

Acknowledgements

We would like to thank George Necula, Matt Welsh, Feng Zhou, and Russ Cox for their helpful contributions. We would also like to thank the Berkeley Millennium group for loaning us the hardware for the benchmarks in this paper. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship, and under the NSF Grant for Millennium, EIA-9802069.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix ATC*, June 2002.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996.

[4] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[5] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002.

[6] D. R. Engler, G. R. Andrews, and D. K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical Report 93-13, Massachusetts Institute of Technology, 1993.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[9] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.

[10] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems, IRIA*, October 1978.

[11] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[12] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[13] J. H. Reppy. Higher-order concurrency. Technical Report 92-1285, Cornell University, June 1992.

[14] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the Telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, 2001.

[15] E. Toernig. Coroutine library source. http://www.goron.de/froese/coro/.

[16] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, , and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[17] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

TCP offload is a dumb idea whose time has come

Jeffrey C. Mogul
Hewlett-Packard Laboratories
Palo Alto, CA, 94304
JeffMogul@acm.org

Abstract

Network interface implementors have repeatedly attempted to offload TCP processing from the host CPU. These efforts met with little success, because they were based on faulty premises. TCP offload *per se* is neither of much overall benefit nor free from significant costs and risks. But TCP offload in the service of very specific goals might actually be useful. In the context of the replacement of storage-specific interconnect via commoditized network hardware, TCP offload (and more generally, offloading the transport protocol) appropriately solves an important problem.

1 Introduction

TCP [18] has been the main transport protocol for the Internet Protocol stack for twenty years. During this time, there has been repeated debate over the implementation costs of the TCP layer.

One central question of this debate has been whether it is more appropriate to implement TCP in host CPU software, or in the network interface subsystem. The latter approach is usually called "TCP Offload" (the category is sometimes referred to as a "TCP Offload Engine," or TOE), although it in fact includes all protocol layers below TCP, as well. Typical reasons given for TCP offload include the reduction of host CPU requirements for protocol stack processing and checksumming, fewer interrupts to the host CPU, fewer bytes copied over the system bus, and the potential for offloading computationally expensive features such as encryption.

TCP offload poses some difficulties, including both purely technical challenges (either generic to all transports or specific to TCP), and some more subtle issues of technology deployment.

In some variants of the argument in favor of TCP offload, proponents assert the need for transport-protocol offload but recognize the difficulty of doing this for TCP, and have proposed deploying new transport protocols that support offloading. For example, the XTP protocol [8] was originally designed specifically for efficient implementation in VLSI, although later revisions of the specification [23] omit this rationale.

To this day, TCP offload has never firmly caught on in the commercial world (except sometimes as a stopgap to add TCP support to immature systems [16]), and has been scorned by the academic community and Internet purists. This paper starts by analyzing why TCP offload has repeatedly failed.

The lack of prior success with TCP offload does not, however, necessarily imply that this approach is categorically without merit. Indeed, the analysis of past failures points out that novel applications of TCP might benefit from TCP offload, but for reasons not clearly anticipated by early proponents. TCP offload does appear to be appropriately suited when used in the larger context in which storage-interconnect hardware, such as SCSI or FiberChannel, is on the verge of being replaced by Ethernet-based hardware and specific upper-level protocols (ULPs), such as iSCSI. These protocols can exploit "Remote Direct Memory Access" (RDMA) functionality provided by network interface subsystems. This paper ends by analyzing how TCP offload (and more generally, offloading certain transport protocols) can prove useful, not as a generic protocol implementation strategy, but as a component in an RDMA design.

This paper is *not* a defense of RDMA. Rather, it argues that the choice to use RDMA more clearly justifies offloading the transport protocol than has any previous application.

2 Why TCP offload is a dumb idea

TCP offload has been unsuccessful in the past for two kinds of reasons: fundamental performance issues, and difficulties resulting from the complexities of deploying TCP offload in practice.

2.1 Fundamental performance issues

Although TCP offload is usually justified as a performance improvement, in practice the performance benefits are either minimized or actually negated, for many reasons:

Limited processing requirements: Processing TCP headers simply doesn't (or shouldn't) take many cycles. Jacobson [11] showed how to use "header

prediction" to process the common case for a TCP connection in very few instructions. The overhead of the TCP protocol *per se* does not justify offloading. Clark *et al.* [9] showed more generally that TCP should not be expensive to implement.

Moore's Law: Adding a transport protocol implementation to a Network Interface Controller (NIC) requires considerably more hardware complexity than a simple MAC-layer-only NIC. Complexity increases time-to-market, and because Moore's Law rapidly increases the performance of general-purpose CPU chips, complex special-purpose NIC chips can fall behind CPU performance. The TOE can become the bottleneck, especially if the vendor cannot afford to utilize the latest fab. (On the other hand, using a general-purpose CPU as a TOE could lead to a poor tradeoff between cost and performance [1].)

Partridge [17] pointed out that the Moore's Law issue could be irrelevant once each NIC chip is fast enough to handle packets at full line rate; further improvements in NIC performance might not matter (except to reduce power consumption). Sarkar *et al.* [21], however, showed that current protocol-offload NIC system products are not yet fast enough. Their results also imply that any extra latency imposed by protocol offload in the NIC will hurt performance for real applications. Moore's Law considerations may plague even "full-line-rate" NICs until they are fast enough to avoid adding much delay.

Complex interfaces to TOEs: O'Dell [14] has observed that "the problem has always been that the protocol for talking to the front-end processor and gluing it onto the API was just as complex (often more so, in fact) as the protocol being `offloaded'." Similarly, Partridge [16] observed that "The idea was that you passed your data over the bus to an NIC that did all the TCP work for you. However, it didn't give a performance improvement because to a large degree, it recreated TCP over the bus. That is, for each write, you had to add a bus header, including context information (identifying the process and TCP connection IDs) and then ship the packet down to the board. On inbound, you had to pass up the process and TCP connection info and then the kernel had to demux the bus unit of data to the right process (and do all that nasty memory alignment stuff to put it into the process's buffer in the right place)." While better approaches are now known, in general TOE designers had trouble designing an efficient host interface.

Suboptimal buffer management: Although a TOE can deliver a received TCP data segment to a chosen location in memory, this still leaves ULP protocol headers mingled with ULP data, unless complex features are included in the TOE interface.

Connection management: The TOE must maintain connection state for each TCP connection, and must coordinate this state with the host operating system. Especially for short-lived connections, any savings gained from less host involvement in processing data packet is wasted by this extra connection management overhead.

Resource management: If the transport protocol resides in the NIC, the NIC and the host OS must coordinate responsibility for resources such as data buffers, TCP port numbers, etc. The ownership problem for TCP buffers is more complex than the seemingly analogous problem for packet buffers, because outgoing TCP buffers must be held until acknowledged, and received buffers sometimes must be held pending reassembly. Resource management becomes even harder during overload, when host OS policy decisions must be supported. None of these problems are insuperable, but they reduce the benefits of offloading.

Event management: Much of the cost of processing a short TCP connection comes from the overhead of managing application-visible events [2]. Protocol offload does nothing to reduce the frequency of such events, and so fails to solve one of the primary costs of running a busy Web server (for example).

Much simpler NIC extensions can be effective: Numerous projects have demonstrated that instead of offloading the entire transport protocol, a NIC can be more simply extended so as to support extremely efficient TCP implementations. These extensions typically eliminate the need for memory copies, and/or offload the TCP checksum (eliminating the need for the CPU to touch the data in many cases, and thus avoiding data cache pollution). For example, Dalton *et al.* [10] described a NIC supporting a single-copy host OS implementation of TCP. Chase *et al.* [7] summarize several approaches to optimizing end-system TCP performance.

These criticisms of TCP offload apply most clearly when one starts with a well-tuned, highly scalable host OS implementation of TCP. TCP offload might be an expedient solution to the problems caused by second-rate host OS implementations, but this is not itself an architectural justification for TOE.

2.2 Deployment issues

Even if TCP offload were justified by its performance, it creates significant deployment, maintenance, and management problems:

Scaling issues: Some servers must maintain huge num-

bers of connections [2]. Modern host operating systems now generally place no limits except those based on RAM availability. If the TOE implementation has lower limits (perhaps constrained by on-board RAM), this could limit system scalability. Scaling concerns also apply to the IP routing table.

Bugs: Protocol implementations have bugs. Mature implementations have fewer bugs, but still require patches from time to time. Updating the firmware of a programmable TOE could be more difficult than updating a host OS. Clearly, non-programmable TOEs are even worse in this respect [1].

Quality Assurance (QA): System vendors must test complete systems prior to shipping them. Use of TOE increases the number of complex components to be tested, and (especially if the TOE comes from a different supplier) increases the difficulty of locating bugs.

Finger-pointing: When a TCP-related bug appears in a traditional system, it is not hard to decide whether the NIC is at fault, because non-TOE NICs perform fairly simple functions. With a system using TCP offloading, deciding whether the bug is in the NIC or the host could be much harder.

Subversion of NIC software: O'Dell has argued that the programmability of TOE NICs offers a target for malicious modifications [14]. This argument is somewhat weakened by the reality that many (if not most) high-speed NICs are already reprogrammable, but the extra capabilities of a TOE NIC might increase the options for subversion.

System management interfaces: System administrators prefer to use a consistent set of management interfaces (UIs and commands). Especially if the TOE and OS come from different vendors, it might be hard to provide a consistent, integrated management interface. Also, TOE NICs might not provide as much state visibility to system managers as can be provided by host OS TCP implementations.

Concerns about NIC vendors: NIC vendors have typically been smaller than host OS vendors, with less sophistication about overall system design and fewer resources to apply to support and maintenance. If a TOE NIC vendor fails or exits the market, customers can be left without support.

While none of these concerns are definitive arguments against TOE, they have tended to outweigh the limited performance benefits.

2.3 Analysis: mismatched applications

While it might appear from the preceding discussion that TCP offload is inherently useless, a more accurate statement would be that past attempts to employ TCP offload were mismatched to the applications in question.

Traditionally, TCP has been used either for WAN networking applications (email, FTP, Web) or for relatively low-bandwidth LAN applications (Telnet, X/11). Often, as is the case with email and the Web, the TCP connection lifetimes are quite short, and the connection count at a busy (server) system is high.

Because these are seen as the important applications of TCP, they are often used as the rationale for TCP offload. But these applications are exactly those for which the problems of TCP offload (scalability to large numbers of connections, per-connection overhead, low ratio of protocol processing cost to intrinsic network costs) are most obvious. In other words, in most WAN applications, the end-host TCP-related costs are insignificant, except for the connection-management costs that are either unsolved or worsened by TOE.

The implication of this observation is that the sweet spot for TCP offload is not for traditional TCP applications, but for applications that involve high bandwidth, low-latency, long-duration connections.

3 Why TCP offload's time has come

Computers generate high data rates on three kinds of channels (besides networks): graphics systems, storage systems, and interprocessor interconnects. Historically, these rates have been provided by special-purpose interface hardware, which trades flexibility and price for high bandwidth and high reliability.

For storage especially, the cost and limitations of special-purpose connection hardware is increasingly hard to justify, in the face of much cheaper Gbit/sec (or faster) Ethernet hardware. Replacing fabrics such as SCSI and Fiber Channel with switched Ethernet connections between storage and hosts promises increased configuration flexibility, more interoperability, and lower prices.

However, replicating traditional storage-specific performance using traditional network protocol stacks would be difficult, not because of protocol processing overheads, but because of data copy costs – especially since host busses are now often the main bottleneck. Traditional network implementations require one or more data copies, especially to preserve the semantics of system calls such as read() and write(). These APIs allow applications to choose when and how data buffers appear in their address spaces. Even with in-kernel applications (such as NFS), complete copy avoidance is not easy.

Several OS designs have been proposed to support traditional APIs and kernel structures while avoiding all unnecessary copies. For example, Brustoloni [4, 5] has explored several solutions to these problems.

Nevertheless, copy-avoidance designs have not been widely adopted, due to significant limitations. For example, when network maximum segment size (MSS)

values are smaller than VM page sizes, which is often the case, page-remapping techniques are insufficient (and page-remapping often imposes overheads of its own.) Brustoloni also points out that "many copy avoidance techniques for network I/O are not applicable or may even backfire if applied to file I/O." [4]. Other designs that eliminate unnecessary copies, such as I/O Lite [15], require the use of new APIs (and hence force application changes). Dalton *et al.* [10] list some other difficulties with single-copy techniques.

Remote Direct Memory Access (RDMA) offers the possibility of sidestepping the problems with software-based copy-avoidance schemes. The NIC hardware (or at any rate, software resident on the NIC) implements the RDMA protocol. The kernel or application software registers buffer regions via the NIC driver, and obtains protected buffer reference tokens called *region IDs*. The software exchanges these region IDs with its connection peer, via RDMA messages sent over the transport connection. Special RDMA message directives ("verbs") enable a remote system to read or write memory regions named by the region IDs. The receiving NIC recognizes and interprets these directives, validates the region IDs, and performs protected data transfers to or from the named regions.[1]

In effect, RDMA provides the same low-overhead access between storage and memory currently provided by traditional DMA-based disk controllers.

(Some people have proposed factoring an RDMA protocol into two layers. A Direct Data Placement (DDP) protocol simply allows a sender to cause the receiving NIC to place data in the right memory locations. To this DDP functionality, a full RDMA protocol adds a remote-read operation: system *A* sends a message to system *B*, causing the NIC at *B* to transfer data from one of *B*'s buffers to one of *A*'s buffers without waking up the CPU at *B*. David Black [3] argues that a DDP protocol by itself can provide sufficient copy avoidance for many applications. Most of the points I will make about RDMA also apply to a DDP-only approach.)

An RDMA-enabled NIC (RNIC) needs its own implementation of all lower-level protocols, since to rely on the host OS stack would defeat the purpose. Moreover, in order for RDMA to substitute for hardware storage interfaces, it must provide highly reliable data transfer, so RDMA must be layered over a reliable transport such as TCP or SCTP [22]. This forces the RNIC to implement the transport layer.

Therefore, offloading the transport layer becomes valuable not for its own sake, but rather because that allows offloading of the RDMA layer. And offloading the RDMA layer is valuable because, unlike traditional TCP applications, RDMA applications are likely to use a relatively small number of low-latency, high-bandwidth transport connections, precisely the environment where TCP offloading might be beneficial. Also, RDMA allows the RNIC to separate ULP data from ULP control (i.e., headers) and therefore simplifies the received-buffer placement problems of pure TCP offload.

For example, Magoutis *et al.* [13] show that the RDMA-based Direct Access File System can outperform even a zero-copy implementation of NFS, in part because RDMA also helps to enable user-level implementation of the file system client. Also, storage access implies the use of large ULP messages, which amortize offloading's increased per-packet costs while reaping the reduced per-byte costs.

Although much of the work on RDMA has focussed on storage systems, high-bandwidth graphics applications (e.g., streaming HDTV videos) have similar characteristics. A video-on-demand connection might use RDMA both at the server (for access to the stored video) and at the client (for rendering the video).

4 Implications for operating systems

Because RDMA is explicitly a performance optimization, not a source of functional benefits, it can only succeed if its design fits comfortably into many layers of a complete system: networking, I/O, memory architecture, operating system, and upper-level application. A misfit with any of these layers could obviate any benefits.

In particular, an RNIC design done without any consideration for the structures of real operating systems will not deliver good performance and flexibility. Experience from an analogous effort, to offload DES cryptography, showed that overlooking the way that software will use the device can eliminate much of the potential performance gain [12]. Good hardware design is certainly not impossible, but it requires co-development with the operating system support.

RDMA aspects requiring such co-development include:

Getting the semantics right: RDMA introduces many issues related to buffer ownership, operation completion, and errors. Members of the various groups trying to designs RDMA protocols (including the RDMA Consortium [19] and the IETF's RDDP Working Group [20]) have had difficulty resolving many basic issues in these designs. These disagreements might imply the lack of sufficiently mature principles underlying the mixed use of remotely- and locally-managed buffers.

OS-to-RDMA interfaces: These interfaces include, for example, buffer allocation; mapping and protection of buffers; and handling exceptions beyond what the RNIC can deal with (such as routing and ARP information for a new peer address).

Application-to-RDMA interfaces: These interfaces

include, for example, buffer ownership; notification of RDMA completion events; and bidirectional interfaces to RDMA verbs.

Network configuration and management: RNICs will require IP addresses, subnet masks, etc., and will have to report statistics for use by network management tools. Ideally, the operating system should provide a "single system image" for network management functions, even though it includes several independent network stack implementations.

Defenses against attacks: an RNIC acts as an extension of the operating system's protection mechanisms, and thus should defend against subversions of these mechanisms. The RNIC could refuse access to certain regions of memory known to store kernel code or data structures, except in narrowly-defined circumstances (e.g., bootstrapping).

Since the RNIC includes a TCP implementation, there will be temptation to use that as a pure TOE path for non-RDMA TCP connections, instead of the kernel's own stack. This temptation must be resisted, because it will lead to over-complex RNICs, interfaces, and host OS modifications. However, an RNIC might easily support certain simple features that have been proposed [5] for copy-avoidance in OS-based network stacks.

5 Difficulties

RDMA introduces several tricky problems, especially in the area of security. Prior storage-networking designs assumed a closed, physically secure network, but IP-based RDMA potentially leaves a host vulnerable to the entire world.

Offloading the transport protocol exacerbates the security problem by adding more opportunities for bugs. Many (if not most) security holes discovered recently are implementation bugs, not specification bugs. Even if an RDMA protocol design can be shown to be secure, this does not imply that all of its implementations would be secure. Hackers actively find and exploit bugs, and an RDMA bug could be much more severe than traditional protocol-stack bugs, because it might allow unbounded and unchecked access to host memory.

RDMA security therefore cannot be provided by sprinkling some IPSec pixie dust over the protocol; it will require attention to all layers of the system.

The use of TCP below RDMA is controversial, because it requires TCP modifications (or a thin intermediate layer whose implementation is entangled with the TCP layer) in order to reliably mark RDMA message boundaries. While SCTP is widely accepted as inherently better than TCP as a transport for RDMA, some vendors believe that TCP is adequate, and intend to ship RDMA/TCP implementations long before offloaded SCTP layers are mature. This paper's main point is not that *TCP* offload is a good idea, but rather that *transport-protocol* offload is appropriate for RNICs. TCP might simply represent the best available choice for several years.

6 Conclusions

TCP offload has been "a solution in search of a problem" for several decades. This paper identifies several inherent reasons why general-purpose TCP offload has repeatedly failed. However, as hardware trends change the feasibility and economics of network-based storage connections, RDMA will become a significant and appropriate justification for TOEs.

RDMA's remotely-managed network buffers could be an innovation analogous to novel memory consistency models: an attempt to focus on necessary features for real applications, giving up the simplicity of a narrow interface for the potential of significant performance scaling. But as in the case of relaxed consistency, we may see a period where variants are proposed, tested, evolved, and sometimes discarded. The principles that must be developed are squarely in the domain of operating systems.

Acknowledgments

I would like to thank David Black, Craig Partridge, and especially Jeff Chase, as well as the anonymous reviewers, for their helpful comments.

References

[1] B. S. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. Tech. Rep. HPL-2001-8, HP Labs, Jan. 2001.

[2] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Annual Technical Conf.*, pages 1–12, New Orleans, LA, June 1998. USENIX.

[3] D. Black. Personal communication, 2003.

[4] J. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proc. INFOCOM '99*, pages 534–542, New York, NY, Mar. 1999. IEEE.

[5] J. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. OSDI-II*, pages 277–291, Seattle, WA, Oct. 1996. USENIX.

[6] J. S. Chase. *High Performance TCP/IP Networking* (Mahbub Hassan and Raj Jain, Editors), chapter 13, *TCP Implementation.* Prentice-Hall. In preparation.

[7] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End-system optimizations for high-speed TCP. *IEEE Communications*, 39(4):68–74, Apr. 2001.

[8] G. Chesson. XTP/PE overview. In *Proc. IEEE 13th Conf. on Local Computer Networks*, pages 292–296, Oct. 1988.

[9] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[10] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: Architectural support for high performance protocols. *IEEE Network Magazine*, 7(4):36-43, 1995.

[11] V. Jacobson. 4BSD TCP header prediction. *Computer Communication Review*, 20(2):13–15, Apr. 1990.

[12] M. Lindemann and S. W. Smith. Improving DES coprocessor throughput for short operations. In *Proc. 10th USENIX Security Symp.*, Washington, DC, Aug. 2001.

[13] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the Direct Access File System. In *Proc. USENIX 2002 Annual Tech. Conf.*, pages 1–14, Monterey, CA, June 2002.

[14] M. O'Dell. Re: how bad an idea is this? Message on TSV mailing list, Nov. 2002.

[15] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Trans. Computer Systems*, 18(1):37–66, Feb. 2000.

[16] C. Partridge. Re: how bad an idea is this? Message on TSV mailing list, Nov. 2002.

[17] C. Partridge. Personal communication, 2003.

[18] J. B. Postel. Transmission Control Protocol. RFC 793, Information Sciences Institute, Sept. 1981.

[19] RDMA Consortium. http://www.rdmaconsortium.org.

[20] Remote Direct Data Placement Working Group. http://www.ietf.org/html.charters/rddp-charter.html.

[21] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: Does hardware support help? In *Proc. 2nd USENIX Conf. on File and Storage Technologies*, pages 231–244, San Francisco, CA, March 2003.

[22] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, Network Working Group, Oct. 2000.

[23] T. Strayer. Xpress Transport Protocol, Rev. 4.0b. XTP Forum, 1998.

Notes

[1] Much of this paragraph was adapted, with permission, from a forthcoming book chapter by Jeff Chase [6].

TCP Meets Mobile Code

Parveen Patel David Wetherall Jay Lepreau Andrew Whitaker
University of Utah and University of Washington

Abstract

This paper argues that transport protocols such as TCP provide a rare domain in which protocol extensibility by untrusted parties is both valuable and practical. TCP continues to be refined despite more than two decades of progress, and the difficulties due to deployment delays and backwards-compatibility are well-known. Remote extensibility, by which a host can ship the transport protocol code and dynamically load it on another node in the network on a per-connection basis, directly tackles both of these problems. At the same time, the unicast transport protocol domain is much narrower than other domains that use mobile code, such as active networking, which helps to make extensibility feasible. The transport level provides a well understood notion of global safety—TCP friendliness—while local safety can be guaranteed by isolation of per-protocol state and use of recent safe-language technologies. We support these arguments by outlining the design of XTCP, our extensible TCP framework.

1 Introduction

TCP was designed over two decades ago and has been evolving ever since. Proposals for changes show no sign of ceasing, as they are driven by changes in the way the network is used and the quest for ever better performance [22, 11, 18, 13, 6, 15, 12, 30, 31, 21, 24, 47, 2, 20, 27, 29, 40, 43, 8, 10, 3, 1, 38, 46, 32, 37]. Yet the process of evolution itself is not simple or painless. Experimentation with a new version of TCP requires that both communication endpoints be upgraded, in the general case. Widespread deployment is needed to unlock the true value of such extensions, which in practice takes years, lowering the value for early adopters and posing a further barrier to change.

These difficulties have resulted in pressure to produce TCP extensions that require upgrades at a single endpoint, even at the expense of efficiency or robustness. For example, both NewReno [18] and SACK [31] improve performance when there are multiple packet losses in one window of data. NewReno is based on a heuristic interpretation of duplicate acknowledgments, and can be deployed for immediate benefit. In contrast, SACK addresses multiple losses by design, has been shown to provide improved performance, and is generally considered the better alternative [17]. The catch is that SACK requires both ends to be upgraded.

An alternative approach is to build remote extensibility mechanisms into TCP itself, so that both end points can be upgraded at once. This would free protocol designers from both the constraints of backwards-compatibility and the deployment barrier. This argument should sound familiar: it is essentially the argument for active networks, which aims to allow new network services to be introduced using mobile code. Yet active networking has not seen widespread deployment; the many reasons include the lack of compelling applications and the technical difficulties of running user-defined code within the network. Furthermore, prior work on extensible operating systems cannot readily be used in this domain due to its lack of support for extending a remote operating system with untrusted protocol code.

In this paper, we put forth the case for XTCP, a remotely extensible version of TCP. We argue that the domain of TCP extensions provides a sweet spot that is well-suited to leverage the mobile code aspect of active networking, without incurring the problems that have hindered active networks. Past and proposed TCP variations demonstrate a clear need for such extensibility: we present an analysis of 27 TCP variants and find that the majority would benefit from remote extensibility. At the same time, compared to the generality of active networks, TCP provides a restricted domain within which the technical challenges can be successfully tackled.

A key challenge in providing this rapid deployment model is to do so without causing security problems. Transport protocol code that comes from sources that are not authoritative—such as the other end of a wide-area connection—should not be trusted. We must ensure that such code cannot compromise the integrity of the host system or consume too many of its resources. Further, standard practice in the networking community is to re-

Author information: Patel, Lepreau: University of Utah, {ppatel,lepreau}@cs.utah.edu; Wetherall, Whitaker: University of Washington, {djw,andrew}@cs.washington.edu.

This work was supported in part by DARPA grants F30602-99-1-0503, F33615-00-C-1696, and a Microsoft Endowment Fellowship.

quire that new transport protocols compete fairly with deployed versions of TCP to ensure that they will not undermine the stability of the network [19]. Thus, to provide a system that is acceptable in practice we must provide this form of network safety. Since extensions to TCP are often undertaken to improve performance, we must also allow new transport extensions to be competitive in performance with hard-coded and manually deployed versions.

Our design makes automatic deployment practical by exploiting TCP's connection phase for in-band signaling of protocol requirements, deferring code distribution to user-mode daemons so that later connections benefit. We use the concept of TCP-friendliness to provide a clear model of network safety, and the recent ECN nonce mechanism [16] to enforce TCP-friendliness without trusting local TCP extensions or any remote parties. To obtain host safety with reasonable impact on performance and the structure of traditional kernels, we exploit TCP's stylized memory allocation and its limited sharing between connections, use a C-like type-safe language [25], and enforce resource limitations. We have focused on TCP to date for concreteness, but expect much of our reasoning to apply to other transports such as DCCP [27] and SCTP [43].

The rest of this paper is organized as follows. In section 2, we develop the case in favor of remote extensibility at the transport layer, while in section 3 we show that such extensibility is achievable by presenting the design of the XTCP framework. In section 4, we conclude with a discussion of the key issues for our future research.

2 The Case for XTCP

2.1 XTCP is Useful

We envision the following scenarios for using XTCP:

1. A "high performance" TCP is installed along with a Web server, and code is pushed to receivers to provide more rapid downloads. Figure 1 illustrates this example scenario using the XTCP extensibility model discussed in section 3.1.

2. A mobile client installs "TCP connection migration" [41] and ships code to the server to allow itself to move.

3. A user installs "TCP nice" [45] to provide background data transfers. No remote code shipping is needed.

To demonstrate that such extensibility is useful, we surveyed TCP extensions and TCP-friendly transports that have been deployed or proposed since congestion control was first introduced in 1988 in TCP Tahoe [22]. We analyzed 27 TCP extensions and classified them into three categories according to which endpoints must be upgraded to gain a benefit, assuming Tahoe as the baseline implementation. The results are shown in Table 1.

We found that the 16 extensions listed in Category 1 require upgrades to both sender and receiver sides to be of value. For some of these extensions, such as TCP connection migration, it is very hard, if not impossible, to benefit by modifying only one endpoint. XTCP provides a clear benefit for these extensions, allowing them to be readily deployed where they otherwise could not.

The five extensions listed in Category 2 can be implemented by upgrading a single endpoint and XTCP would not seem to benefit them directly. However, all of these designs have the potential to become either more robust or effective if both ends can be upgraded and the new functionality split freely between the sender and the receiver. For example, NewReno could become SACK, and TCP Vegas [11] could use receiver timings to more accurately estimate queuing delay [35]. That is, these extensions are constrained to some extent by the pressure of backwards-compatibility; XTCP would alleviate this pressure.

Finally, the remaining six extensions in Category 3 require changes to only one endpoint. For example, both the fast recovery modification to the sender-side TCP and TCP-Nice are transparent to the receiver. For these protocols, XTCP can still provide a useful kernel upgrade mechanism by allowing third-party software authors to write and remotely install TCP extensions.

In summary, the majority of the extensions we studied benefit from the XTCP model of remote extensibility, and many would be difficult to deploy without it.

2.2 XTCP is Practical

The second issue we consider is technical feasibility, since XTCP requires the use of mobile code and is similar in spirit to the challenging domain of active networks [44]. The key insight and difference between XTCP and active networks is that TCP (or more generally, unicast transport protocols) is a much narrower domain in which to provide extensibility. This enables several key simplifications that increase our confidence in being able to build an effective solution.

First, XTCP provides extensibility at a much coarser granularity than active networks: per connection at endpoints rather than per packet at routers. This permits a simpler approach to upgrades, where extensions are signaled in-band, at connection setup time, and code is transferred in the background.

Second, there exists a clear model of "global" or network safety for XTCP: a connection should not be able to send faster than a TCP-friendly transport. In fully general active networks, there is no clear limit to

Category	Extensions
1. *Require* both endpoints to change	**1. Connection migration**: Migrating live TCP connections [41], **2. SACK**: Selective acks [31], **3. D-SACK**: Duplicate SACK [21], **4. FACK**: Forward acks [30], **5. RFC 1323**: TCP extensions for high-speed networks [24], **6. TCPSAT**: TCP for satellite networks [4], **7. ECN**: Explicit congestion notification [36], **8. ECN nonce**: Detects masking of ECN signals by the receiver or network [16], **9. RR-TCP**: Robustly handles packet reordering [47], **10. WTCP**: TCP for wireless WANs [37], **11. The Eifel algorithm**: Detection of spurious retransmissions [29], **12. T/TCP**: TCP for transactions [10], **13. TFRC**: Equation-based TCP-friendly congestion control [20], **14. DCCP**: New transport protocol with pluggable congestion control [27], **15. SCTP**: Transport protocol support for multi-homing, multiple streams etc., between endpoints [43], **16. RAP**: Rate adaptive TCP-friendly congestion control [38]
2. *Could benefit more* if both endpoints could change	**1. NewReno**: Approximation of SACK from sender side [18] **2. TCP Vegas**: A measurement-based adaptive congestion control [11], **3. TCP Westwood**: Congestion control using end-to-end rate estimation [46], **4. Karn/Partridge algorithm**: Retransmission backoff and avoids spurious RTO estimates due to retransmission ambiguity [26], **5. Congestion manager**: A generic congestion control layer [7]
3. *Require* only one endpoint to change	**1. Header prediction**: Common case optimization on input path [23], **2. Fast recovery**: Faster recovery from losses [42], **3. Syn-cookies**: Protection against SYN-attacks [8], **4. Limited transmit**: Performance enhancement for lossy networks [2], **5. Appropriate byte-counting**: Counting bytes instead of segments for congestion control [1], **6. TCP nice**: TCP for background transfers [45]

Table 1: Classification of TCP extensions, assuming TCP Tahoe [22] as the baseline version.

the network-wide resources that can be expended on a packet, and even if there were, multiple extensible routers would need to cooperate to enforce the limit. Furthermore, recent advances in network safety mechanisms allow enforcement of a rate-limiting model. Compliance can be checked by the local XTCP using a variant of the ECN nonce mechanism, without trusting either extension code or remote hosts.

Third, even in terms of local safety—protecting the resources of the local host—XTCP affords simplifications compared to active networks. There is limited sharing between TCP connections in the kernel, which translates into simpler protection models and eases the task of termination, should code need to be unloaded for any reason. Recent advances in safe language technology also contribute to XTCP's practicality. Our design leverages Cyclone [25], a type-safe variant of C, to obtain host protection while providing relatively straightforward reuse of existing C-based transport protocols, familiarity to system programmers, and acceptable performance.

A fourth simplification is that XTCP can leverage a large body of past TCP extensions upon which to base its extension API. Active networks, on the other hand, had no such agreed set of applications and hence it tended towards generality. In contrast, XTCP aims to do one thing, and to do it well.

3 Design

The basic approach of XTCP is to download transport extensions directly into the operating system kernel. To guarantee safety, XTCP uses type-safety to achieve memory protection and restricts extensions to a resource-safe API, called the XTCP API, summarized in Table 2. Extensions are invoked in response to specific system events, such as packet input/output and timers, and are allowed to read and write IP datagrams. To ensure network safety, trusted XTCP network access functions attach IP headers to outgoing datagrams and limit the sending rate of a transport to that of a TCP-friendly transport. Extensions are allowed to register timers, manipulate packet buffers, and interact with the sockets layer in a safe manner. This functionality is sufficient to implement a range of transport protocols, including conventional TCP. In this section, we focus on three key aspects of XTCP's design: *connection setup and code distribution*, *network safety*, and *host safety*.

3.1 Connection Setup and Code Distribution

Before a transport extension can be used, the code must be distributed to both connection endpoints. XTCP accomplishes this by interposing on normal TCP connection setup, as shown in Figure 1. Suppose host A wants to communicate with host B using a transport extension. Host A's first packet (either a SYN or a SYN-ACK) includes a special TCP option, which includes a hash of the desired transport extension's code. If host B has already loaded the extension, the current connection is established using the requested protocol. If B has not loaded the extension, it issues a request to A for the new protocol code, while using the default TCP to establish the current connection.

Asynchronously "at leisure," a user-level daemon on A transfers the code to B's daemon (connection 2 in the figure). B either compiles source code, requests a trusted server to do so, or verifies the authenticity of object code,

The XTCP API Categories
1. Protocol management
xtcp_load_proto(proto_sw)
xtcp_unload_proto(proto_handle)
2. Sockets layer
xtcp_sowakeup(socket)
xtcp_sbapppend(socket, seg)
xtcp_isdisconnecting(socket)
xtcp_sobind(socket)
3. Connection management callback functions
xtcp_attach(socket)
xtcp_connect(socket, remote-endpoint, state)
xtcp_abort(socket)
xtcp_accept(socket, remote-endpoint, state)
4. TCP-friendly network access
xtcp_ack(end_seqno, nonce)
xtcp_ack_sum(end_seqno, noncesum)
xtcp_nack(seqno)
xtcp_net_sendack(segment)
xtcp_net_send(segment, seqno)
xtcp_net_resend(segment, new_seqno, old_seqno)
5. Runtime support
xtcp_gettick()
xtcp_seg_alloc(proto_handle)
xtcp_timer_reset(proto_handle, callout)
xtcp_get_rtentry(proto_handle, dst_ip_addr)

Table 2: The initial XTCP API exports a set of 67 functions. The major groups and sample functions are listed.

loads it into the kernel, and makes it available for subsequent network connections. This signaling and code distribution scheme only benefits later connections---often substantially later, since the entire process could take many seconds or perhaps even minutes, since it must be done at low priority to mitigate DoS attacks. Its effectiveness relies on the pattern of TCP connections common in today's Internet, in which hosts make repeated TCP connections to a particular peer.

This connection setup procedure has several features we believe are important to practical deployment. It is fully backwards-compatible with conventional TCP, imposes minimal control latency on the connection that is used to bootstrap a new extension, minimally invades the kernel software architecture, and retains TCP's three-way packet exchange for compatibility with existing level 4 "middleboxes" like firewalls, NAT boxes, and other proxies. However, note that this backwards-compatible procedure is limited to extensions that use TCP-like handshakes; new transport protocols are free to use their own connection protocol after bootstrapping.

Our current design provides extensibility at the granularity of an entire TCP implementation. This coarse-grained design provides complete flexibility and is practical in most domains: our measurements show that a full TCP implementation, with comments and headers, takes 85K bytes of compressed source code; Cyclone source code should not expand it at all [25]. If certified object code is transferred instead of source, its size is smaller. We measured 20K of x86 object code, which Cyclone should expand by at most 20%.

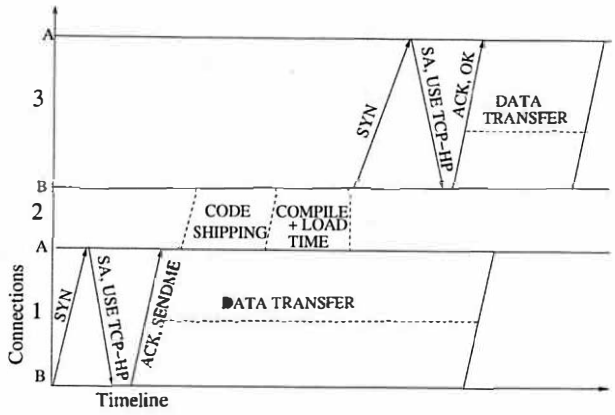


Figure 1: An example scenario. In connection 1, server A requests use of high-performance TCP (TCP-HP), causing client B to ask for the code. At application level in connection 2, server A sends it to client B. In a later, unrelated connection 3, A and B use TCP-HP.

We have not discussed the multitude of potential policy issues regarding which extensions to invoke or accept from peers. Briefly, our design uses three sources to select extensions: application-provided socket options, host-wide configuration options, and the code distribution and policy server that communicates with other such servers. The policy servers might "rate" protocols [39], could form a web of (partial) trust, and in fact could represent the beginnings of an Internet "knowledge plane" [14]. However, we believe only the simplest policies need be implemented for our design to function well.

3.2 Network Safety

The network safety goal of XTCP is to require new transport protocols to compete fairly with deployed versions of TCP to ensure that they will not undermine the stability of the network, as recommended in RFC 2914 [19]. Currently, XTCP achieves this by limiting extensions to a TCP-friendly sending rate, as modeled by the TCP rate equation [33]. This equation gives an upper bound on allowable sending rate of a TCP-friendly flow in terms of *packet size*, *loss event rate* and *round trip time*.

XTCP must compute values of these parameters without trusting the local transport or remote endpoint. This is essential to prevent new transports from compromising the rate-checking mechanism by indirectly inflating the allowable sending rate. For this purpose, XTCP adapts

the recently proposed ECN nonce mechanism [16].

The ECN nonce mechanism is based on placing a random one-bit value in the IP header of outgoing packets. The nonce bit (or a sum of nonce bits over many packets) is later used as a proof of acknowledgment. The extensions must inform XTCP of packet arrival and loss events in terms of per-packet sequence numbers; these numbers appear as an extra argument to the network send function. Upon receiving an acknowledgment, the extension reports the sequence number and nonce, using one of the acknowledgment functions shown in Table 2. Packet drops are indicated by using negative-acknowledgment functions; also, XTCP assumes that a packet has been lost if the nonce is incorrect or a timeout period has expired. This information is sufficient for XTCP to estimate the packet size, loss event rate, and round trip time parameters needed by the TCP rate equation.

A crucial feature of the above mechanism is that XTCP remains independent of transport header formats, permitting arbitrary header modifications by new extensions.

3.3 Host Safety

Host safety in the face of untrusted code is achieved through principles of isolation and resource control similar to those used in safe language-based operating systems, such as KaffeOS [5], but simplified by the constrained memory allocation and sharing behavior of TCP. Memory safety is achieved by using Cyclone, a type-safe dialect of C [25]. The type-safety of Cyclone prevents memory corruption, and its compatibility with C makes it easier and more efficient to interface with traditional kernels than other safe languages, such as Java or OCaml.

Extensions are never allowed to share memory with other extensions, grab system locks, or disable interrupts. Therefore, asynchronous termination can be safely achieved by terminating all connections of a misbehaving extension. This tractable notion of termination allows us to use traditional run-time techniques to bound memory usage and inexpensive timer interrupt-based checking to bound CPU usage.

4 Open Issues and Conclusion

In this paper, we have argued that TCP, and more generally unicast transport protocols, present a unique domain in which remote extensibility by untrusted parties is both valuable for users and technically feasible. We presented the design of XTCP, our framework for achieving this extensibility. We have implemented a prototype of XTCP in the FreeBSD kernel and ported TCP NewReno [18] and TCP SACK [31] to it. Our initial experience with XTCP's performance, safety characteristics, and ease of porting existing protocols has been encouraging [34].

There are several open issues that we expect to tackle as we gain experience with the system. First, we are using the TCP-friendly rate equation to provide network safety. This equation governs the steady-state transfer rate as a function of loss, and to date it has mostly been used to build other transports such as TFRC. We use it online to police TCP extensions. This means we must determine appropriate timescales on which to apply it. The timescales must be long enough to avoid false alarms, but short enough to prevent abusive transports from crowding out compliant transports.

Second, a key issue is whether our XTCP API will prove sufficient to support a wide variety of TCP extensions, including those currently unimagined. The need to repeatedly revise the XTCP API would defeat its purpose. We believe that our static API in conjunction with mobile code will prove sufficiently general because it fulfills a key need of extensions: the ability to change packet formats, allowing the sender and receiver to exchange and process new information. Our current API directly supports 18 of the 21 extensions listed in Categories 1 and 2 in table 1. Three extensions cannot be supported because they are not TCP-friendly. However, only experience will tell whether XTCP can fulfill its twin goals of supporting a large fraction of useful transport extensions while guaranteeing host and network safety.

A related issue is whether we can design an OS-independent XTCP API such that the same code can be compiled across operating systems. This will further expedite protocol deployment by allowing developers to "write-once" and bypass all OS-specific issues.

Finally, it is interesting to consider the granularity of extensions. We began by partially designing a fine-grained extensibility model, but switched to shipping TCP implementations in their entirety. This coarse-grained model provides complete flexibility, avoids feature interaction, is simple, and appears practical, given the observed modest code sizes. A fine-grained composable TCP implementation, analogous to that in Prolac [28] and FoxNet [9], would reduce the size of transported code and host memory consumption, but could lead to a less flexible extension model. We suspect that simple will win.

Acknowledgments

We thank the anonymous reviewers whose comments helped improve an earlier version of this paper. We are grateful to Tim Stack, Mike Hibler, Rob Ricci, and John Regehr for implementation, evaluation, and editing help.

References

[1] M. Allman. TCP Congestion Control with Appropriate Byte Counting. *IETF Internet Draft draft-allman-tcp-abc-04.txt*, Oc-

tober 2002.

[2] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. *RFC 3042*, 2001.

[3] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. *RFC 3390*, 2002.

[4] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP Over Satellite Channels using Standard Mechanisms. *RFC 2488*, January 1999.

[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *OSDI*. USENIX Association, Oct. 2000.

[6] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. *15th International Conference on Distributed Computing Systems*, 1995.

[7] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, pages 175–187, 1999.

[8] D. J. Bernstein and E. Schenk. TCP Syn Cookies. 1996, 2002; http://cr.yp.to/syncookies.html.

[9] E. Biagioni. A Structured TCP in Standard ML. In *ACM SIGCOMM*, 1994.

[10] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. *RFC 1644*, 1994.

[11] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE JSAC*, 13(8), 1995.

[12] K. Brown and S. Singh. M-TCP: TCP for Mobile Cellular Networks. *Computer Communication Review*, 1997.

[13] F. Chengpeng. *TCP Veno: End-To-End Congestion Control Over Heterogeneous Networks*. PhD thesis, Chinese University of Hong Kong, 2001.

[14] D. D. Clark, C. Partridge, and J. C. Ramming. A Knowledge Plane for the Internet, Feb. 2003. Unpublished report; earlier versions available at http://www.isi.edu/braden/know-plane/.

[15] R. Durst, G. Miller, and E. Travis. TCP Extensions for Space Communications. In *ACM MobiCom*, Nov. 1996.

[16] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signaling. In *IEEE ICNP*, November 2001.

[17] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM Computer Communication Review*, 26(3), Jul 1996.

[18] S. Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 2582*, 1999.

[19] S. Floyd. Congestion Control Principles. *RFC 2914*, September 2000.

[20] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.

[21] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. *RFC 2883*, 2000.

[22] V. Jacobson. Congestion Avoidance and Control. *SIGCOMM*, 1988.

[23] V. Jacobson. 4BSD Header Prediction. *ACM Computer Communication Review*, April 1990.

[24] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC 1323*, 1992.

[25] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Chene, and Y. Wang. Cyclone: A Safe Dialect of C. *USENIX Annual Technical conference, Monterey, CA*, June 2002.

[26] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, 1991.

[27] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram Congestion Control Protocol (DCCP), October 2002. http://www.icir.org/kohler/dccp/.

[28] E. Kohler, F. Kaashoek, and D. Montgomery. A Readable TCP in the Prolac Protocol Language. In *SIGCOMM*, pages 3–13, 1999.

[29] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communication Review*, 30(1), January 2000.

[30] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *SIGCOMM*, 1996.

[31] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. *RFC 2018*, 1996.

[32] J. Nagle. Congestion Control in IP/TCP. *RFC 896*, January 1984.

[33] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM*, 1998.

[34] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading Transport Protocols using Untrusted Mobile Code. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003. To appear.

[35] V. Paxson. End-to-end Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3), 1999.

[36] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168*, 2001.

[37] K. Ratnam and I. Matta. WTCP: An Efficient Transmission Control Protocol for Networks with Wireless Links. *Proc. Third IEEE ISCC*, 1998.

[38] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet. In *INFOCOM (3)*, 1999.

[39] R. Ricci and J. Lepreau. Active Protocols for Agile Censor-Resistant Networks. In *8th HotOS*. IEEE Computer Society, May 2001.

[40] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *ACM Computer Communication Review*, 28(4), October 1998.

[41] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *6th MobiCom*, 2000.

[42] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *RFC 2001*, January 1997.

[43] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. *RFC 3309*, 2002.

[44] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[45] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *OSDI*, 2002.

[46] R. Wang, M. Valla, M. Y. Sanadidi, and M. Gerla. Adaptive Bandwidth Share Estimation in TCP Westwood. In *IEEE Globecom*, November 2002.

[47] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. TR 006, International Computer Science Institute (ICSI), Berkeley, California, USA, July 2002.

Exploiting the Synergy between Peer-to-Peer and Mobile Ad Hoc Networks

Y. Charlie Hu, Saumitra M. Das, and Himabindu Pucha
Purdue University
West Lafayette, IN 47907
{ychu, smdas, hpucha}@purdue.edu

Abstract

We argue that there exists a synergy between peer-to-peer (p2p) overlay networks for the Internet and mobile ad hoc networks (MANETs) connecting mobile nodes communicating with each other via multi-hop wireless links – both share the key characteristics of self-organization and decentralization, and both need to solve the same fundamental problem, that is, how to provide connectivity in a decentralized, dynamic environment. We propose Dynamic P2P Source Routing (DPSR), a new routing protocol for MANETs that exploits the synergy between p2p and MANETs for increased scalability. By integrating Dynamic Source Routing (DSR) and a proximity-aware structured p2p overlay routing protocol, DPSR limits the number of the source routes that each node has to discover and rediscover to $O(logN)$, while retaining all the attributes of DSR for dealing with the specifics of ad hoc networks. This is in contrast to the maximum of N source routes each node has to maintain in DSR. Thus DPSR has the potential to be more scalable than previous routing protocols for MANETs, such as DSR and AODV.

In addition to being a network layer multi-hop routing protocol, DPSR simultaneously implements a distributed hash table (DHT) in MANETs; it implements the same functionalities as CAN, Chord, Pastry, and Tapestry, which can be exposed to the applications built on top of it via a set of common p2p APIs.

1 Introduction

A peer-to-peer (p2p) overlay network consists of a dynamically changing set of nodes connected via the Internet (i.e., IP). A mobile ad hoc network (MANET) consists of mobile nodes communicating with each other using multi-hop wireless links. P2p overlays and MANETs share the key characteristics of self-organization and decentralization. These common characteristics lead to further similarities between the two types of networks: (1) Both have a flat and frequently changing topology, caused by node join and leave in p2p overlays and MANETs and additionally terminal mobility of the nodes in MANETs; and (2) Both use hop-by-hop connection establishment. Per-hop connections in p2p are typically via TCP links with physically unlimited range, whereas per-hop connections in MANETs are via wireless links, limited by the radio transmission range.

The common characteristics shared by p2p overlays and MANETs also dictate that both networks are faced with the same fundamental challenge, that is, to provide connectivity in a decentralized, dynamic environment. Thus, there exists a synergy between these two types of networks in terms of the design goals and principles of their routing protocols; both p2p and MANET routing protocols have to deal with dynamic network topologies due to membership changes or mobility.

We argue that a promising research direction in networking is to exploit the synergy between p2p overlay and MANET routing protocols to design better routing protocols for MANETs. As a supporting example, in this paper, we apply a recent advancement in p2p overlay networks, i.e., proximity-aware structured p2p overlay routing protocols, to routing in MANETs, and propose a new routing protocol that promises to be more scalable than previous MANET routing protocols.

The primary challenge with using a p2p routing protocol in MANETs is the fact that p2p overlays in the wired Internet rely on the IP routing infrastructure to perform hop-by-hop routing between neighboring nodes in the overlays, whereas such an infrastructure does not exist in MANETs. The obvious idea of overlaying a p2p network (protocol) on top of a multi-hop routing protocol can be inefficient, as it is difficult to exploit the interactions between the two protocols. Instead, our proposed new routing protocol for MANETs, Dynamic P2P Source Routing protocol (DPSR), seamlessly integrates functions performed by p2p overlay routing protocols operating in a logical namespace and by MANET routing protocols operating in a physical namespace. Specifically, DPSR integrates Dynamic Source Routing (DSR) [8] and Pas-

try [16], a proximity-aware structured p2p overlay routing protocol. The key idea of the integration is to bring the structured p2p routing protocol to the network layer of MANETs via a one-to-one mapping between the IP addresses of the mobile nodes and their nodeIds in the namespace, and replacing each routing table entry which used to store a (nodeId, IP address) pair with a (nodeId, source route) pair. With this integration, DPSR limits the number of the source routes that each node has to discover and rediscover to $O(logN)$, while retaining all the attributes of DSR for dealing with the specifics of ad hoc networks, i.e., due to wireless transmissions. Compared to the maximum of N source routes each node has to maintain in DSR, the bounded number of source routes managed by each node in DPSR has the potential to make DPSR much more scalable than previous routing protocols for MANETs, such as DSR and AODV.

2 Background

DPSR is based on the DSR protocol for MANETs and a structured p2p overlay routing protocol, Pastry. In the following, we give a brief overview of DSR and Pastry.

2.1 DSR

DSR [8] is a representative multi-hop routing protocol for ad hoc networks. It is based on the concept of source routing in contrast to hop-by-hop routing. It includes two mechanisms, *route discovery* and *route maintenance.*

Route discovery is the process by which a source node discovers a route to a destination for which it does not already have a route in its cache. The process broadcasts a ROUTE REQUEST packet which is flooded across the network in a controlled manner. In addition to the address of the initiator of the request and the target of the request, each ROUTE REQUEST packet contains a route record, which records the sequence of hops taken by the ROUTE REQUEST packet as it propagates through the network. ROUTE REQUEST packets use sequence numbers to prevent duplication. The request is answered by a ROUTE REPLY packet from the destination node. To reduce the cost of route discovery, each node maintains a cache of source routes that have been learned or overheard, which it uses aggressively to limit the propagation range of ROUTE REQUESTS.

When a route is in use, the route maintenance procedure monitors the operation of the route and informs the sender of any routing errors. A host detects transmission of corrupted or lost packets via the link-level acknowledgment frame defined by IEEE 802.11, or by a passive acknowledgment, i.e., after a host sends a packet to the next hop, it overhears whether the next hop forwards the packet further along the path. If the route breaks due to a link failure, the detecting host sends a ROUTE ERROR packet to the source which upon receiving it, removes all routes in the host's cache that use the hop in error.

Optimizations suggested for DSR for reducing the overhead of route discovery include: (1) overheard and forwarded routing information are cached to reduce the frequency of route discovery invocations; (2) cached routes are used to generate replies to ROUTE REQUESTS to limit the propagation of ROUTE REQUESTS; and (3) ROUTE REPLY storms caused by nodes replying from their caches are prevented by delaying each reply by a period proportional to the number of hops to the destination. This also increases the probability that the source receives the shortest route first.

Optimizations suggested for DSR for improving the effectiveness of route maintenance include: (1) every node helps to maintain shorter routes by sending a gratuitous ROUTE REPLY if it knows of a shorter route to the destination than the one used in an overheard packet; (2) each node always attempts to salvage a data packet that has caused a ROUTE ERROR; (3) ROUTE ERROR packets received by a source node are piggybacked on its next route request to ensure increased spreading of information about stale routes; and (4) ROUTE ERROR packets that are forwarded or eavesdropped on are used to invalidate locally cached routes that contain the hop in error.

Comparison studies of DSR with other proposed routing protocols for MANETs [3, 6] have shown that DSR exhibits good performance at all mobility rates.

2.2 Pastry

Pastry [16] is one of several proximity-aware structured p2p routing protocols [4]. Although it is chosen for the design of DPSR in this paper, other structured p2p protocols such as CAN [15], Chord [17], and Tapestry [18] could potentially be used as well.

In a Pastry network, each node has a unique, uniform, randomly assigned nodeId in a circular 128-bit identifier space. Given a message and an associated 128-bit key, Pastry reliably routes the message to the live node whose nodeId is numerically closest to the key.

In a Pastry network consisting of N nodes, a message can be routed to any node in less than $\log_{2^b} N$ steps on average (b is a configuration parameter with typical value 4), and each node stores only $O(logN)$ entries, where each entry maps a nodeId to the associated node's IP address. Specifically, a Pastry node's routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $(2^b - 1)$ entries each. Each of the $(2^b - 1)$ entries at row n of the routing table refers to a node whose nodeId shares the first n digits with the present node's nodeId, but whose $(n+1)$th digit has one of the $(2^b - 1)$ possible values other than the $(n+1)$th digit in the present node's nodeId. In addition

to a routing table, each node maintains a leaf set, consisting of $L/2$ nodes with numerically closest larger nodeIds, and $L/2$ nodes with numerically closest smaller nodeIds, relative to the present node's nodeId. L is an even integer parameter with typical value 16. In each routing step, the current node forwards a message to a node whose nodeId shares with the message key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the current nodeId. If no such node is found in the routing table, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the current nodeId. Such a node must exist in the leaf set unless the nodeId of the current node or its immediate neighbor is numerically closest to the key.

Node join An arriving node with a newly chosen nodeId X initializes its state by contacting a nearby node A (according to the proximity metric) and asking A to route a special message with X as the key. This message is routed to the existing node Z whose nodeId is numerically closest to X. X then obtains the leaf set from Z, and the ith row of the routing table from the ith node encountered along the route from A to Z. Finally, X announces its presence to the initial members of its leaf set and routing table, which in turn update their own leaf sets and routing tables.

3 Key Concepts

Although DSR is one of the leading MANET routing protocols, ad hoc networks constructed using DSR are still far from scalable when compared to the "fixed" Internet. [1] Simulations performed in ad hoc network protocol studies such as [3, 6] have been limited to networks of up to 100 nodes and a small number of network connections (source-destination pairs). The fundamental reason for the limited scalability of such protocols is that any ad hoc network routing protocol has to pay a high overhead dealing with the dynamic network topology and the shared medium access of wireless communication (e.g., for a 100 node network using DSR, the ratio of routing overhead to data packets for moderate to high mobility ranges from 2:1 to 10:1). Specifically, the size of the route cache in a DSR node is proportional to the number of distinct destination nodes to which it has to send messages, and thus is potentially as high as N, the size of the network. Note that the memory required to store such routes is not a scalability concern. Rather, it is the overhead required to discover and rediscover these many routes that limits the scalability of DSR.

[1] We note that position-based routing protocols which rely on global position systems can be more scalable than topology-based protocols such as DSR.

Destination	Source Route
$< nodeId_x >$	$< S_i ... S_x >$

Table 1. A DPSR routing table or leaf set entry.

In contrast, in structured p2p overlay networks such as Pastry, each node maintains $O(\log N)$ routing state, independent of the number of different destinations that node has to send messages to. This suggests that a promising approach to improving the scalability of DSR is to limit the size of the routing state each node has to maintain by leveraging efficient structured p2p routing protocols. In the rest of the paper, we propose DPSR as one way of integrating DSR and Pastry.

4 DPSR Design

Like DSR, DPSR is proposed as a network layer protocol. Message destinations and nodes are addressed using IP addresses. DPSR adds a level of indirection to multi-hop routing in MANETs by assigning nodeIds from a circular name space to nodes in the MANET. A prefix-based routing scheme similar to Pastry is then employed to route data packets in the name space. Prefix-based routing has been shown to provide low delay stretch and other useful proximity properties as demonstrated by Pastry [4, 5].

4.1 Basic Design

NodeId assignment DPSR assigns unique nodeIds to nodes in a MANET as is done in Pastry. NodeIds are generated as the secure hashing (SHA-1) of the IP addresses of the hosts. Since the number of nodes in MANETs is small, i.e., in the order of hundreds, this ensures that with very high probability the nodeIds are unique.

Node state The structures of the routing table and the leaf set stored in each DPSR node are similar to those in Pastry. The only difference lies in the content of each leaf set and routing table entry. Since there is no underlying routing infrastructure in MANETs, each entry in a DPSR leaf set or a routing table stores the route to reach the designated nodeId, as shown in Table 1. As in Pastry, each routing table entry for a given node is chosen such that it is physically closer to that node than other choices for that routing table entry. This is achieved via a similar node joining process as in Pastry.

Routing Routing in the basic DPSR design is the same as in Pastry: a message key is first generated by hashing the message's destination IP address, and the message is routed using Pastry's prefix-based routing procedure. In DPSR, since both message keys and nodeIds are hashed from IP addresses, an exact match between a message

key and the destination node's nodeId is expected. In other words, a message will be delivered to the destination node whose nodeId matches the message key, if that destination node is reachable via the wireless links. The only difference between DPSR and Pastry routing is that each hop in the DPSR network is a multi-hop source route, whereas each hop in the Pastry network is a multi-hop Internet route.

Node join The DPSR node joining process is similar to that of Pastry. The only difference is in constructing the contents of the routing table and leaf set entries: each entry in a DPSR routing table or a leaf set stores the source route to a DPSR node, while an entry in Pastry simply stores the IP address of a Pastry node. In both cases, network proximity is taken into consideration when choosing the best node for each routing table entry.

Node failure or out of reach Node failure is again handled similarly as in Pastry. In Pastry, if a node is not reachable, it is presumed to have failed. To replace a failed node in the leaf set, its neighbor in the nodeId space contacts the live node with the largest index on the side of the failed node, and asks that node for its leaf set. This set only partly overlaps with the present node's leaf set. Among these new nodes, the appropriate ones are then chosen and inserted into the leaf set. In DPSR, a node could become unreachable via a source route for two reasons: it or other node(s) along the route has either crashed, or has moved out of the range of its adjacent nodes along the route. In either case, a route discovery for that node is invoked on-demand. If the route discovery still does not find a new route to the unreachable node, that node, if present in the leaf set, is replaced in a similar way as in Pastry.

4.2 Optimizations

The basic design of DPSR inherits all of the optimizations on route discovery and maintenance used by the DSR protocol (see Section 2.1). A number of additional optimizations are unique to the DPSR routing structures and operations.

Use of indirectly received source routes There are three ways a DPSR node can discover source routes: (i) via explicit route discovery; (ii) via overhearing routes in messages sent between neighboring nodes; (iii) via forwarded source routes. In the basic operations of DPSR, a node always chooses the shortest route explicitly discovered for each entry. As an optimization, for every route indirectly received, i.e., via (ii) and (iii), a node checks whether the route is a better candidate than the current corresponding entry in the leaf set and the routing table. If so, the new route replaces the old entry. This optimization thus constantly discovers fresh and low proximity routes for the leaf set and routing table entries.

Routing table and leaf set as route caches In addition to the "prefix-based view" of the routing table, or the "neighbor-node view" of the leaf set, the two routing structures can be viewed as two caches of source routes, similar to the route cache in DSR. This allows the use of *implicit source routes* to destinations, as in the DSR protocol. An implicit source route is a source route embedded in a normal source route. For example, an explicit source route $A \rightarrow B \rightarrow C \rightarrow D$ contains two implicit routes, $A \rightarrow B$ and $A \rightarrow B \rightarrow C$. The implicit source routes can be exploited to optimize the DPSR routing procedure. To send a data packet, it first searches all implicit source routes in the routing table and the leaf set for an exact match between the message key and the destination nodeIds. If this initial search returns a source route, DPSR uses it directly. Otherwise, the original DPSR lookup algorithm, same as Pastry's, is executed to return the source route to the next DPSR hop. In addition, these implicit routes can be used to populate newly created leaf set and routing table entries, for example, when a new node joins.

3-D routing table and 2-D leaf set To further extend the above idea of using leaf sets and routing tables as route caches, leaf sets can be extended to 2-D and routing tables extended to 3-D, i.e., each entry in a leaf set and a routing table contains a vector of N_r routes, where N_r is a configuration parameter. For each explicit and implicit route in each directly or indirectly received route, a node checks whether there is an exact match between the route's destination nodeId and some entry in the leaf set. If so, the route is inserted into the leaf set entry. In addition, the route is also inserted into the unique entry in the routing table, i.e., the one that matches the longest prefix with the nodeId of the route's destination. Obviously, the optimal choice of the number of backup routes N_r depends on the tradeoff between the availability of routes and their freshness. To maintain the freshness of the cached routes, an approximate FIFO replacement policy similar to that used in the DSR cache is used.

The above 3-D routing table and 2-D leaf set have two benefits: (i) They effectively increase the sizes of "route caches" by a factor of N_r, increasing the probability of finding an implicit route in routing data packets. (ii) They potentially reduce the need for route maintenance. If the shortest route, based on the hop count, in a leaf set or routing table entry is broken, instead of performing a route discovery to the destination node, the node can switch to use the shortest route among the backups in the same entry.

4.3 Discussions

4.3.1 Design Alternatives to the Pastry operations

A unique characteristic of MANETs, shared medium access, suggests several design alternatives to the original Pastry protocol operations. In a shared medium, packet delivery can be unreliable due to collisions in transmission. On the other hand, overhearing of packets transmitted by neighboring nodes can be used for routing state maintenance.

The Pastry joining algorithm requires the transmission of many critical messages each of which when lost, would cause restarting the entire joining process. The algorithm assumes a low probability of packet loss and a low cost of message transmission in the network. Both of these assumptions do not hold for wireless ad hoc networks. Additionally, the join algorithm (directly inherited from Pastry) in it's final stage requires the joining node to discover routes to all members of its leaf set and routing table, each of which may require a flooding, for a total of $L + O(logN)$ floodings. This suggests a potentially more efficient joining process in which the joining node simply floods the entire network once, and a selected subset of nodes, e.g., the potential candidates for the leaf set and the routing table entries, send replies back to the flooding node.

The Pastry routing table maintenance algorithm is designed to preserve the locality of routing table entries in the presence of network dynamics. The algorithm involves periodic communication with nodes in a subset of routing table entries and a subsequent comparison of the proximity of the exchanged routing table entries with the node's own. However, in MANETs, such periodic communication violates the on-demand nature of DSR and thus may incur high overhead. Proximity probing is a very high overhead exercise in MANETs if the route to the probed node needs to be discovered. The nature of the shared medium access of MANETs provides an efficient alternative. A node can use overhearing of routes to maintain locality of its routing table entries. In fact, the nature of the overhearing process guarantees that the routes overheard are from the physically nearby nodes. Continuous updates to routing table entries using overhearing can make DPSR resilient to degradation of routing table quality due to mobility. The cost of this operation is just the power consumed to operate the network access device in promiscuous mode.

4.3.2 Scalability

In MANETs, two notions of scalability are of interest: (1) up till what network sizes a reasonable data packet delivery ratio can be maintained, and (2) for a fixed-size network, how large the routing overhead is for a fixed packet delivery ratio. The lower the routing overhead, the more network bandwidth is available for sending data packets. The two notions, however, are inter-related. If a network is not as congested in delivering a fixed volume of data packets, it can be scaled up further.

We qualitatively compare the routing overhead of DPSR with DSR. As described in Section 2.1, using DSR, depending on the number of distinct destinations a node sends messages to, each node needs to maintain up to N routes in a MANET of N nodes. In contrast, using DPSR, the number of routes each node needs to maintain is limited to $O(logN)$, independent of the number of different destinations that node has to send messages to. The exact tradeoff between DPSR and DSR is more complicated since both discover and rediscover routes on-demand. But to the first order of approximation, DPSR is expected to incur less overhead than DSR when each node communicates with on average over $O(logN)$ other nodes (one-to-many).[2] The reduced routing overhead allows more data packets which compete for accessing the shared medium to be delivered successfully.

Since DPSR routes packets through several overlay hops, whereas DSR takes the direct path, if queuing delay is discounted, the routing delay using DPSR is expected to be longer than using DSR. However, the delay stretch, defined as the delay going through the overlay hops divided by the delay via a single DSR source route, is expected to be within a factor of two. With a random uniform distribution of nodeIds and node locations in the 2-D proximity space, the lengths of consecutive overlay hops in routing a message in DPSR increase exponentially by a factor of $2^{b/2}$, since the number of nodes matching each additional digit decreases by a factor of 2^b. Since the last hop dominates, and the earlier hops are directionless, i.e., they are equally likely to move towards or away from the destination node, the expected delay stretch is bounded by $\frac{1}{1-1/2^{b/2}}$, and thus is small than 2 for $b > 1$. Furthermore, this delay can be reduced whenever implicit routes are found and used to deliver data packets directly to their destination nodes.

In summary, under the first notion of scalability, if there are many nodes that communicate with multiple other nodes, we expect DPSR to scale up to a larger network size than DSR from lower routing overhead. As the network size increases, however, the scalability of both DSR and DPSR will be limited by the lengths of the source routes, because the longer the source routes, the more likely they will break. Under the second notion of scalability, DPSR is expected to deliver more packets than DSR for one-to-many communication patterns and perform comparably when a node communicates on average with a few other nodes.

[2] For example, many p2p applications exhibit such traffic patterns.

5 Other Related Work

PeerNet [7] is a p2p-based network layer similar to DPSR in that both aim at improving the scalability of routing protocols by bringing the p2p concept from the application layer down to the network layer. However, PeerNet focuses on dynamic networks with pockets of wireless connectivities interconnected with wired lines, whereas DPSR focuses on wireless ad hoc networks.

In addition to DSR, AODV [14], DSDV [13], and TORA [12] also belong to the category of topology-based multi-hop ad hoc routing protocols which assume no knowledge of the mobile nodes' positions. Such position information typically requires the assistance of global positioning systems. In contrast, position-based protocols forward packets based on the physical positions of nodes. These include "flooding-based" such as LAR [10] and DREAM [1], "graph-based" such as RGD [2], and "geographic-based" such as GPSR [9]. Among these, geographic forwarding approaches route packets based on only local decisions, and thus have less overhead and are more scalable. GLS [11] is a scalable distributed location service that can be combined with geographic forwarding to construct large ad hoc networks.

Acknowledgment

We thank Peter Druschel, Dave Johnson, and the anonymous reviewers for their helpful comments. This work was supported in part by an NSF CAREER award (ACI-0238379) and a Honda Initiation Grant 2002.

References

[1] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward. A distance routing effect algorithm for mobility (DREAM). In *Proc. ACM MOBICOM'98.*

[2] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proc. DialM'99.*

[3] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proc. ACM MOBICOM'98.*

[4] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in distributed hash tables. In *Proc. FuDiCo'02.*

[5] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical report, Technical report MSR-TR-2002-82, 2002.

[6] S. R. Das, C. E. Perkins, and E. M. Royer. Performance comparison of two on-demand routing protocols for ad hoc networks. In *Proc. IEEE INFOCOM'00.*

[7] J. Eriksson, M. Faloutsos, and S. Krishnamurthy. Peernet: Pushing peer-to-peer down the stack. In *IPTPS'03.*

[8] D. B. Johnson and D. A. Maltz. *Dynamic Source Routing in Ad Hoc Wireless Networks.* Kluwer Academic, 1996.

[9] B. Karp and H. Kung. GPSR: Greedy perimeter stateless terminode routing. In *Proc. ACM MOBICOM'00.*

[10] Y.-B. Ko and N. H. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Proc. ACM MOBICOM'98.*

[11] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A Scalable Location Service for Geographic Ad Hoc Routing. In *Proc. ACM MOBICOM'00.*

[12] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proc. IEEE INFOCOM'97.*

[13] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proc. ACM SIGCOMM'94.*

[14] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proc. WMCSA'99.*

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM'01.*

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. MIDDLEWARE'01.*

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM'01.*

[18] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Scheduling and Simulation: How to Upgrade Distributed Systems

Sameer Ajmani Barbara Liskov
Massachusetts Institute of Technology
Laboratory for Computer Science
{ajmani,liskov}@lcs.mit.edu

Liuba Shrira
Brandeis University
Computer Science Department
liuba@cs.brandeis.edu

Abstract

Upgrading the software of long-lived distributed systems is difficult. It is not possible to upgrade all the nodes in a system at once, since some nodes may be down and halting the system for an upgrade is unacceptable. This means that different nodes may be running different software versions and yet need to communicate, even though those versions may not be fully compatible. We present a methodology and infrastructure that addresses these challenges and makes it possible to upgrade distributed systems automatically while limiting service disruption.

1 Introduction

Long-lived distributed systems, like server clusters, content distribution networks, peer-to-peer systems, and sensor networks, require changes (upgrades) in their software over time to fix bugs, add features, and improve performance. These systems are large, so it is impractical for an administrator to upgrade nodes manually (e.g., via remote login). Instead, upgrades must propagate automatically, but the administrator may still require control over the order and rate at which nodes upgrade to avoid interrupting service or to test an upgrade on a few nodes. Thus, upgrades may happen slowly, and there may be long periods of time when some nodes are upgraded and others are not. Nonetheless, the system as a whole should continue to provide service.

The goal of our research is to support automatic upgrades for such systems and to enable them to provide service during upgrades. Earlier approaches to automatically upgrading distributed systems [9–11, 17, 18, 22] or distributing software over networks [1–6, 8, 15, 25] do little to ensure continuous service during upgrades. The Eternal system [27], the Simplex architecture [24], and Google [14] enable specific kinds of systems to provide service during upgrades, but they do not provide general solutions.

An automatic upgrade system must:

- propagate upgrades to nodes automatically
- provide a way to control *when* nodes upgrade
- enable the system to provide service when nodes are running different versions
- provide a way to preserve the persistent state of nodes from one version to the next

To address these requirements, our approach includes an *upgrade infrastructure*, *scheduling functions*, *simulation objects*, and *transform functions*.

The *upgrade infrastructure* is a combination of centralized and distributed components that enables rapid dissemination of upgrade information and flexible monitoring and control of upgrade progress.

Scheduling functions (SFs) are procedures that run on nodes and tell them when to upgrade. SFs can implement a variety of upgrade scheduling policies.

Simulation objects (SOs) are adapters that allow a node to behave as though it were running multiple versions simultaneously. Unlike previous approaches that propose similar adapters [13, 23, 27], ours includes correctness criteria to ensure that simulation objects reflect node state consistently across different versions. These criteria require that some interactions made via SOs must fail; we identify when such failure is necessary and, conversely, when it is possible to provide service between nodes running different versions.

Transform functions (TFs) are procedures that convert a node's persistent state from one version to the next. Our contribution is to show how TFs interact with SOs to ensure that nodes upgrade to the correct new state.

Our approach takes advantage of the fact that long-lived systems are *robust*. They tolerate node failures: nodes are prepared for failures and know how to recover to a consistent state. This means that we can model a node upgrade as a soft restart. Robust systems also tolerate communication problems: remote procedure calls may fail, and callers know how to compensate for such failures. This means that we can use a failure response when calls occur at inopportune times, e.g., when a node is upgrading or when a node's simulation object is unable to carry out the requested action.

The rest of the paper is organized as follows. Section 2 presents our upgrade model and assumptions. Section 3 describes the upgrade infrastructure; Section 4, scheduling functions; Section 5, simulation objects; and Section 6, transform functions. Section 7 presents correctness criteria, and Section 8 concludes.

2 Model and Assumptions

We model each node as an object that has an identity and a state; we assume there is just one object per node but the model can be extended to handle multiple objects. Objects are fully-encapsulated; inter-object interaction is by means of method calls. Systems based on remote procedure calls [26] or remote method invocations [21] map easily to this model; extending the model to message-passing [19] is future work.

A portion of an object's state may be persistent. Objects are prepared for failure of their node: when the node recovers, the object reinitializes itself from the persistent portion of its state.

Each node runs a top-level class—the class that implements its object. We assume class definitions are stored in well-known repositories and define the full implementation of a node, including its subcomponents and libraries. Different nodes are likely to run different classes, e.g., clients run one class, while servers run another.

Our approach defines upgrades for entire systems, rather than just for individual nodes. A *version* defines the software for *all* the nodes in the system. An upgrade moves the system from one version to the next. We expect upgrades to be relatively rare, e.g., they occur less than once a month. Therefore, the common case is when all nodes are running the same version. We also expect that before an upgrade is installed, it is thoroughly debugged; our system is not intended to providing a debugging infrastructure.

An upgrade identifies the classes that need to change by providing a set of *class upgrades*: ⟨*old-class, new-class, TF, SF, past-SO, future-SO*⟩. *Old-class* identifies the class that is now obsolete; *new-class* identifies the class that is to replace it. TF is a *transform function* that generates the new object's persistent state from that of the old object. SF is a *scheduling function* that tells a node when it should upgrade. *Past-SO* and *Future-SO* are classes providing *simulation objects*. *Past-SO*'s object implements old-class's behavior by calling methods on the new object (i.e., it provides backward compatibility); *Future-SO*'s object implements new-class's behavior by calling methods on the old object (i.e., it provides forward compatibility). An important feature of our approach is that the upgrade designer only needs to understand two versions: the new one and the preceding one.

Sometimes new-class will implement a subtype of old-class, but we do not assume this. When the subtype relationship holds, no past-SO is needed, since new-class can handle all calls for old-class. Often, new-class and old-class will implement the same type (e.g., new-class just fixes a bug or optimizes performance), in which case neither a past-SO nor a future-SO is needed.

We assume that all nodes running the old-class must switch to the new-class. Eventually we may provide a filter that restricts a class upgrade to only some nodes belonging to the old-class; this is useful, e.g., to upgrade nodes selectively to optimize for environment or hardware capabilities.

3 Infrastructure

The upgrade infrastructure consists of four kinds of components, as illustrated in Figure 1: an *upgrade server*, an *upgrade database*, and per-node *upgrade layers* and *upgrade managers*.

A logically centralized *upgrade server* maintains a *version number* that counts how many upgrades have been installed in the past. An upgrade can only be defined by a trusted party, called the *upgrader*, who must have the right credentials to install upgrades at the upgrade server. When a new upgrade is installed, the upgrade server advances the version number and makes the new upgrade available for download. We can extend this model to allow multiple upgrade servers, each with its own version number.

Each node in the system is running a particular version, which is the version of the last upgrade installed on that node. A node's *upgrade layer* labels outgoing calls made by its node with the node's version number. The upgrade layer learns about new upgrades by querying the upgrade server and by examining the version numbers of incoming calls.

When an upgrade layer hears about a new version, it notifies the node's *upgrade manager (UM)*. The UM downloads the upgrade for the new version from the upgrade server and checks whether the upgrade contains a class upgrade whose old-class matches the node's current class. If so, the node is affected by the upgrade. Otherwise, the node is unaffected and immediately advances its version number.

If a node is affected by an upgrade, its UM fetches the appropriate class upgrade and class implementation from the upgrade server. The UM verifies the class upgrade's authenticity then installs the class upgrade's future-SO, which lets the node support (some) calls at the new version. The node's upgrade layer dispatches incoming calls labeled with the new version to the future-SO.

The UM then invokes the class upgrade's scheduling function, which runs in parallel with the current version's software, determines when the node should upgrade, and signals the UM at that time. The scheduling function may access a centralized *upgrade database* to coordinate the upgrade schedule with other nodes and to enable human operators to monitor and control upgrade progress.

In response to the scheduling signal, the UM shuts down the current node software, causing it to persist

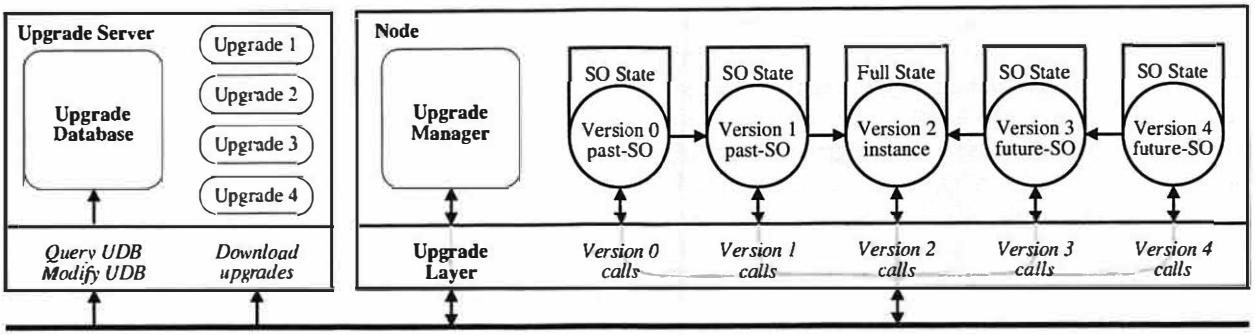


Figure 1: *An example system that consists of an upgrade server and a single system node. Arrows indicate the direction of (remote) method calls. The upgrade server stores the upgrade database and publishes upgrades for versions 1 through 4. The node is running version 2, has a chain of past-SOs to support versions 0 and 1, and has a chain of future-SOs to support versions 3 and 4.*

its state. The UM then installs the new class implementation and runs the transform function to convert the node's persistent state to the representation required by new version. The UM then discards the future-SO and installs the past-SO, which lets the node continue to support the previous version. Finally, the UM starts the new version's software, which recovers from the newly-transformed persistent state.

4 Scheduling Functions

Scheduling functions (SFs) are procedures defined by the upgrader that tell nodes when to upgrade. Unlike existing systems that coordinate upgrades centrally [4,15,25], SFs run on the nodes themselves. This lets SFs respond quickly to changing environments, e.g., to avoid upgrading a replica if another one fails. This approach can also reduce communication and so save energy in resource-constrained systems.

Here are examples of upgrade schedules and SFs:

Upgrade eagerly. The SF signals immediately. This schedule is useful to fix a critical bug.

Upgrade gradually. The SF decides whether to signal by periodically flipping a coin. This schedule can avoid causing too many simultaneous node failures and recoveries, e.g., in a peer-to-peer system.

Upgrade one-replica-at-a-time. The SF signals if its node has the lowest IP address among its non-upgraded replicas. This schedule is useful for replica groups that tolerate only a few failures [7,27].

Upgrade after my servers upgrade. The SF signals once its node's servers have upgraded. This schedule prevents a client node from calling methods that its servers do not yet fully support.

Upgrade all nodes of class C1 before nodes of class C2. The SF queries the upgrade database to determine when to signal its UM. This schedule imposes a partial order on node upgrades.

Upgrade only nodes 1, 2, and 5. This schedule lets the upgrader test an upgrade on a few nodes [25].

Many other schedules are possible, e.g., to avoid disturbing user activity or to avoid creating blind spots in sensor networks.

In general, we cannot predict what parts of a node's state an SF might use to implement its policy. Instead, we provide SFs with read-only access to *all* of a node's state via privileged observers. Restricting SFs to read-only access prevents them from violating the node's specification by mutating its state.

An SF may also need to know the versions and classes of other nodes. The upgrade database (UDB) provides a generic, central store for such information. Upgrade layers (ULs) store their node's class and version in the UDB after each upgrade. SFs can query the UDB to implement globally-coordinated schedules, and the upgrader can query the UDB to monitor upgrade progress. ULs also exchange this information with other ULs and cache it, so SFs can query ULs for information about recently-contacted nodes. The upgrader can define additional upgrade-specific tables in the UDB, e.g., a list of nodes that are authorized to upgrade. The upgrader can modify these tables to control upgrade progress.

The main challenge in designing scheduling functions is ensuring that they behave correctly. Since SFs control the rate at which node upgrades occur, they can affect a system's availability, fault-tolerance, and performance. We are investigating ways to reason about SF correctness and their system-wide effects.

5 Simulation Objects

Simulation objects (SOs) are defined by the upgrader to enable communication between nodes running differ-

ent versions. This is necessary when nodes upgrade at different times, since nodes running older versions may make calls on nodes running newer versions, and vice versa. It is important to enable simulation in both directions, because otherwise a slow upgrade can partition upgraded nodes from non-upgraded ones (since calls between those nodes will fail). Simulation also simplifies software development by allowing implementors to write their software as if every node in the system were running the same version.

SOs are wrappers: they delegate (most of) their behavior to other objects. This means that SOs are simpler to implement than full class implementations, but they are also slower than full implementations and may not be able to implement full functionality (as discussed in Section 7). If a new version does not admit good simulation, the upgrader may choose to use an eager upgrade schedule (as discussed in Section 4) and avoid the use of SOs altogether—but the upgrader must bear in mind that an eager schedule can disrupt service.

An upgrader defines two simulation objects for each version, a *past-SO* and a *future-SO*. A *past-SO* implements an old version by calling methods on the object of the next newer version; thus, a chain of past SOs can support many old versions. It is installed when a node upgrades to a new version and is discarded when the infrastructure determines (by consulting the UDB) that it is no longer needed.

A *future-SO* implements a new version by calling methods on the previous version; like past-SOs, future-SOs can be chained together to support several versions. A future-SO is installed when a node learns of a new version and can be installed "on-the-fly" when a node receives a call at a version newer than its own. A future-SO is removed when its node upgrades to the new version.

At a given time, a node may contain a chain of past-SOs and a chain of future-SOs, as depicted in Figure 1. An SO may call methods on the next object in the chain; it is unaware of whether the next object is the current object or another SO. When a node receives a call, its upgrade layer dispatches the call to the object that implements the version of that call. The infrastructure ensures that such an object always exists by dynamically installing future-SOs and by only discarding past-SOs for dead versions.

Simulation objects may contain state and may use this state to implement calls. SOs must automatically recover their state after a node failure. When an SO is installed, it must initialize its state. Past-SOs initialize their state from the old version's persistent state, as depicted in Figure 2. Future-SOs initialize their state without any input.

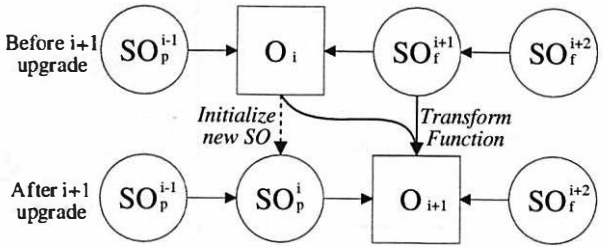


Figure 2: *State transforms for upgrading from version i to version i+1. Squares represent the states of the current version; circles represent the states of simulation objects.*

6 Transform Functions

Transform functions (TFs) are procedures defined by the upgrader to convert a node's persistent state from one version to the next. In previous systems [11, 13, 17], TFs converted the old object into a new one whose representation (a.k.a. "rep") reflected the state of the old one at the moment the TF ran. Our system extends this approach to allow the TF to also access the future-SO created for its version, as illustrated in Figure 2. The TF must then produce a new-class object whose state reflects both the state of the old object and the state of the future-SO. The upgrader can simplify the TF by making the future-SO stateless; then the TF's input is just the old version's state.

In systems that enable nodes to recover their persistent state from other nodes, the TF may be able to simply discard a node's state and rely on state recovery to restore it. This requires that state transfer work correctly between nodes running different versions (e.g., using SOs) and that the scheduling function allow enough time between node upgrades for state transfer.

Previous systems provide tools to generate TFs automatically [16, 20, 27]. We believe such tools can be useful to generate simple TFs and SOs, but creating complex TFs and SOs will require human assistance.

7 Correctness Criteria

This section presents informal correctness criteria for simulation objects. We describe the criteria in the context of a node running version i, O_i, with a single past-SO, SO_p^{i-1}, and a single future-SO, SO_f^{i+1}. We assume atomicity, i.e., calls to a node run in some serial order.

We assume that each version $i+1$ has a specification that describes the behavior of its objects. In addition we require that the specification explain how version $i+1$ is related to the previous version i. This explanation can be given in the form of a *mapping function*, MF_{i+1}, that maps the abstract state of O_i to that of O_{i+1}.

We have the obvious criteria: SO_f^{i+1}, O_i, and $SO_p^{i\ 1}$ must each satisfy their version's specification. In the case of O_i we expect "full compliance," but in the case of the SOs, calls may fail when necessary. One of the main questions we are trying to answer is, when is failure necessary?

There can be multiple clients of a node, and these clients may be running different versions. This means that calls to different versions can be interleaved. For example, a call to O_i (made by a client at version i) may be followed by a call to SO_f^{i+1} (made by a client at version $i+1$), which may be followed by a call to $SO_p^{i\ 1}$ (made by a client running version i 1), and so on. We want this interleaving to make sense.

Also, clients running different versions may communicate about the state of a node. A client at version i may use (observe or modify) the state of the node via O_i, and a client at version $i+1$ may use the state of the node via SO_f^{i+1}, and the two clients may communicate about the state of the node out-of-band. We want the state of the node to appear consistent to the two clients.

7.1 Future SOs

This section discusses the correctness criteria for SO_f^{i+1}. We need to understand what each method of SO_f^{i+1} is allowed to do. The choices are: fail, access/modify the state of O_i, or access/modify the state of SO_f^{i+1}.

When going from version i to $i+1$, some state of O_i is reflected in O_{i+1}, and some is forgotten. We can view the abstract state of O_i as having two parts, a dependent part D_i and an independent part I_i, and the abstract state of O_{i+1} as having two parts, D_{i+1} and I_{i+1}. These parts are defined by MF_{i+1}: MF_{i+1} ignores I_i, uses D_i to produce D_{i+1}, and trivially initializes I_{i+1}.

Now we can describe the criteria for SO_f^{i+1}. A call to SO_f^{i+1} uses (observes or modifies) D_{i+1}, I_{i+1}, or both. Calls that use I_{i+1} execute directly on SO_f^{i+1}'s rep. However, calls that use D_{i+1} must access O_i, or else clients at versions i and $i+1$ may see inconsistencies.

For example, suppose O_i and O_{i+1} are web servers, and O_{i+1} adds support for comments on web pages. MF_{i+1} produces O_{i+1}'s pages from O_i's pages: D_{i+1} = D_i = the pages. O_{i+1}'s comments for each page are independent of O_i's state, i.e., I_{i+1} = the comments. Calls to SO_f^{i+1} to add or view comments can be implemented by accessing SO_f^{i+1}'s rep, where information about comments is stored, but calls that access the pages must be delegated to O_i. This way we ensure, e.g., that a modification of a page made via a call to SO_f^{i+1} will be observed by later uses of O_i.

Thus we have the following condition:

1. Calls to SO_f^{i+1} that modify or observe D_{i+1} must be implemented by calling methods of O_i.

This condition ensures that modifications made via calls to SO_f^{i+1} are visible to users of O_i, and that modifications made via calls to O_i are visible to users of SO_f^{i+1}.

However, there is a problem here: sometimes it is not possible to implement calls on D_{i+1} by delegating to O_i. Suppose O_i is an Archive (a set that can only grow) and O_{i+1} is a Cache (a set that can grow and shrink). Then, D_{i+1} = D_i = the set. SO_f^{i+1} cannot implement removals by delegating to O_i, because O_i does not support removals. SO_f^{i+1} could support removals by keeping track of removed elements in its own rep and "subtracting" these elements when calls observe D_{i+1}, but this creates an inconsistency between the states visible to clients at version i and $i+1$. To prevent such inconsistencies, we require:

2. Calls to SO_f^{i+1} that use D_{i+1} but that cannot be implemented by calling methods of O_i must fail.

So calls on SO_f^{i+1} for removals must fail. SO_f^{i+1} can still implement additions and fetches by calling O_i.

These conditions disallow caching of mutable O_i state in SO_f^{i+1}. Caching of immutable state is allowed since no inconsistency is possible.

7.2 Past SOs

When O_i upgrades to O_{i+1}, a past-SO, SO_p^i, is created to handle calls to version i. We can apply all the above criteria to SO_p^i by substituting SO_p^i for SO_f^{i+1}, D_i for D_{i+1}, and O_{i+1} for O_i. In addition, SO_p^i must initialize its state from I_i, so that calls to SO_p^i that access I_i reflect the effects of previous calls to O_i.

7.3 Transform Functions

TF_{i+1} produces a rep for O_{i+1} from that of O_i, i.e., it is the concrete analogue of MF_{i+1}. Where TF_{i+1} differs is in how it produces the concrete analogue of I_{i+1}: rather than initializing it trivially (as MF_{i+1} does), TF_{i+1} initializes I_{i+1} from the rep of SO_f^{i+1}. This ensures that calls to O_{i+1} that access I_{i+1} reflect the effects of previous calls to SO_f^{i+1}.

8 Future Work

We believe the design sketched above is a good starting point for supporting automatic upgrades for distributed systems, but plenty of work remains to be done. Here are some of the more interesting open problems:

- Formalize correctness criteria for scheduling functions, simulation objects, and transform functions.

- Provide support for nodes that communicate by message-passing rather than by RPC or RMI.
- Provide support for multiple objects per node.
- Investigate ways to run transform functions lazily, so that a node can upgrade to the next version quickly and add additional information to its representation as needed.
- Investigate ways to recover from upgrades that introduce bugs. One possibility is to use a later upgrade to fix an earlier, broken one. This requires a way to undo an upgrade, fix it, then somehow "replay" the undone operations [12].
- Investigate ways to allow the upgrade infrastructure itself to be upgraded.

We are currently implementing a prototype of our upgrade infrastructure for RPC-based systems [26]. We plan to use the prototype to evaluate upgrades for several systems, including Chord and NFS.

Acknowledgments

This research is supported by NSF Grant IIS-9802066 and by NTT. The authors thank George Candea, Alan Donovan, Michael Ernst, Anjali Gupta, Chuang-Hue Moh, Steven Richman, Rodrigo Rodrigues, Emil Sit, and the anonymous reviewers for their helpful comments.

References

[1] APT HOWTO. `http://www.debian.org/doc/manuals/apt-howto/`.

[2] Cisco Resource Manager. `http://www.cisco.com/warp/public/cc/pd/wr2k/rsmn/`.

[3] EMC OnCourse. `http://www.emc.com/products/software/oncourse.jsp`.

[4] Marímba. `http://www.marimba.com/`.

[5] Red Hat up2date. `http://www.redhat.com/docs/manuals/RHNetwork/ref-guide/up2date.html`.

[6] Rsync. `http://www.rsync.org/`.

[7] Windows 2000 clustering: Performing a rolling upgrade. `http://www.microsoft.com/windows2000/techinfo/planning/incremental/roll%upgr.asp`. 2000.

[8] Managing automatic updating and download technologies in Windows XP. `http://www.microsoft.com/WindowsXP/pro/techinfo/administration/manageau%toupdate/default.asp`. 2002.

[9] J. P. A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. 2001.

[10] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th Intl. Conf. on Configurable Dist. Systems*, 1998.

[11] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983. Also as MIT Laboratory for Computer Science Technical Report 303.

[12] A. Brown and D. A. Patterson. Rewind, repair, replay: Three R's to dependability. In *10th ACM SIGOPS European Workshop*, 2002.

[13] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 1991.

[14] S. Ghemawat. Google, inc., personal comm., 2002.

[15] R. S. Hall et al. An architecture for post-development configuration management in a wide-area network. In *Intl. Conf. on Dist. Computing Systems*, 1997.

[16] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *SIGPLAN Conf. on Programming Language Design and Implementation*, 2001.

[17] C. R. Hofmeister and J. M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report CS-TR-3119, University of Maryland, College Park, 1993.

[18] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), 1990.

[19] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Proc. 2nd Intl. Symposium on Operating Systems*, volume 13, 1979.

[20] B. S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1), 2000.

[21] Sun Microsystems. Java RMI specification, 1998.

[22] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, 2000.

[23] T. Senivongse. Enabling flexible cross-version interoperability for distributed services. In *Intl. Symposium on Dist. Objects and Applications*, 1999.

[24] L. Sha, R. Rajkuman, and M. Gagliardi. Evolving dependable real-time systems. Technical Report CMS/SEI-95-TR-005, CMU, 1995.

[25] M. E. Shaddock, M. C. Mitchell, and H. E. Harrison. How to upgrade 1500 workstations on Saturday, and still have time to mow the yard on Sunday. In *Proc. of the 9th USENIX Sys. Admin. Conf.*, 1995.

[26] R. Srinivasan. RPC: Remote procedure call specification version 2. RFC 1831, Network Working Group, 1995.

[27] L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live upgrades of CORBA applications using object replication. In *IEEE Intl. Conf. on Software Maintenance*, 2001.

Development Tools for Distributed Applications

Mukesh Agrawal Srinivasan Seshan
Carnegie Mellon University

Abstract

The emergence of the global Internet has dramatically broadened and changed the computing landscape. In particular, much of the value in contemporary computing systems derives from networked applications. Prominent examples include e-mail, Usenet news, the World Wide Web, and the many varieties of peer-to-peer networks.

However, the number of successful, large-scale, truly distributed, applications is exceedingly small. We argue that a major reason for this is that tools and other facilities available to aid the developers of these applications are inadequate. We propose a life-cycle for these applications. identify challenges that must be met to make the model viable, and detail our initial work towards meeting these challenges.

1 Introduction

The Internet is not living up to its potential. While the Web has been a tremendous success, providing millions of non-technical users with convenient access to information and the ability to perform transactions on-line, the number of truly distributed applications that have succeeded on the Internet is shockingly small. Even those few applications that support interaction among clients, such as chat rooms, auctions, and file-sharing services, require all operations to pass through a common server. As the success of content distribution networks (CDNs) and peer-to-peer (P2P) applications have shown, there is clearly a great demand for large-scale distributed applications. The major barrier to supporting these, and even richer, applications on the Internet is the difficulty of designing, building, testing, and maintaining distributed applications using the tools that comprise the state-of-the-art today.

We can draw a parallel between the complex task of product development and computer application development. In general, manufacturers explicitly manage the life-cycle of their products. Moreover, they typically have specific tools to support different life-cycle phases, e.g., monitoring tools and statistical packages for quality control, and specialized CAD tools for product design. We believe that the analogous life-cycle for an application would have five stages: the design stage, the implementation stage, the testing stage, the deployment and operation stage, and the maintenance and evolution stage. While good tools exist for the life-cycle stages of traditional non-networked applications (e.g., debuggers, profilers and logging tools), no such tools exist for distributed applications. The goal of our work is to provide a tool chain that supports each of the stages in the life-cycle of Internet applications.

Supporting the life-cycle stages of applications in traditional distributed environments has received a great deal of attention in the past. For example, a wide variety of tools are available for traditional distributed systems. These range from simple communication libraries such as MPI (for scientific computing) to comprehensive environments such as Corba (for enterprise applications). However, these tools target smaller scale, mostly closed environments, which are fundamentally different from the Internet.

Recent efforts have begun to address these same challenges in the Internet context. For example, efforts in DHTs [1] and self-organized overlays [3] are developing a collection of building blocks that help in the implementation of large P2P applications. Similarly, simulation tools like ns-2 [6] and open testbeds like Emulab [10] and Planetlab [7] have provided excellent platforms for the comparison of different designs choices. However, the research community has largely overlooked the later stages of the life-cycle – specifically, testing, deployment and evolution of these applications. Additionally [1], possibilities for integration of tools from different life-cycle stages are as yet unexploited. For example, the models of system behavior generated during the design stage may prove useful in debugging errant behavior during operation.

In this paper, we describe some of the challenges in

[1] We thank Margo Seltzer for this observation.

addressing the needs of distributed applications in the later life-cycle stages. Our initial work in supporting distributed applications has concentrated on the problem of maintenance and evolution – specifically the problem of upgrading a distributed application and possibly rolling back an upgrade. We describe the challenges in addressing this problem as well as some of our initial solutions in Section 2. In Section 3, we discuss some of the issues in our next area of focus – testing and debugging deployments of distributed applications. We summarize our observations and conclude in Section 4.

2 Maintenance and Evolution Stage: Software Upgrade/Rollback

An important capability that our distributed application life-cycle requires is the ability to upgrade to new software versions, and when necessary, to revert to previous software rapidly and easily. We refer to this as the software upgrade/rollback problem. In this section, we explore the challenges in addressing this problem and describe part of the design space of possible solutions.

Broadly, upgrades can be classified as *synchronous* or *asynchronous*. A synchronous upgrade is one in which the software on all nodes must be upgraded near-simultaneously. An asynchronous upgrade, in contrast, does not require coordination amongst nodes of the distributed application. Synchronous upgrades make more demands on the upgrade/rollback system than asynchronous upgrades.

An example of a distributed application that might use asynchronous upgrades is a web server cluster. The prototypical architecture for a web server cluster might consist of a load-balancer, a number of identical web servers, and a back-end database. Because the web servers are, in some sense, replicas, the unavailability of a single web server node does not greatly impact the overall service. Also, the web servers do not need to run the same software release in order to maintain the correctness of the overall service. Accordingly, upgrade and rollback need not be coordinated amongst the web server nodes and can be done asynchronously.

Routing applications are examples of distributed applications that may require synchronous upgrades. In contrast to the web server cluster, the nodes are not replicas, so the unavailability of a single node will force the unavailability of any resources unique to that node. In addition, the operation of the routing nodes is not independent. They must cooperate in order to provide service to their clients. Thus, we must either design upgrades to be backwards compatible, or we must coordinate upgrade (and rollback) amongst the routing nodes.

2.1 Related Work

There are two pieces of related work on upgrading large scale distributed systems that are worth noting: the work on upgrading the Internet routing infrastructure [9] and work on upgrading classes in object-oriented databases [5].

The Internet routing infrastructure can be viewed as a large distributed application. As many networking researchers have bemoaned, the difficulty of upgrading or incorporating new functionality into the Internet infrastructure has significantly limited the deployment of new techniques. This difficulty is the result of both a design that does not accommodate automatic deployment of new functionality and the distributed ownership of the Internet (making it difficult to reach consensus about upgrades). The Active Network [9] community spent many years attempting to address these shortcomings with unfortunately little success. However, we believe some of the important lessons from this work and the deployment of new protocols in the Internet do carry over to the area of upgrading distributed applications.

Liskov et al.'s work [5] focuses on upgrading classes in object oriented databases, where it is essential to preserve object state across upgrades. Note that our problem differs significantly from the problem considered by this effort. In our problem, we assume that the distributed application does not require persistent state. Under this assumption, which we believe will hold for many distributed applications, upgrade and rollback can be handled with significantly less developer effort.

2.2 Challenges/Requirements for Upgrade/Rollback

Based on the discussion above, a solution to the upgrade/rollback problem requires specific capabilities:

1. An upgrade/rollback solution should not require excessive change from existing processes for the distribution and installation of software.

2. A solution must enable software rollback with minimal operator intervention. Ideally, when an operator initiates the rollback procedure, the system automatically reverts all necessary state, such as program files, configuration files, etc.

3. The upgrade and rollback procedures should operate quickly.

4. The solution should minimize the down time due to upgrades and rollbacks. Note that this requirement can be relaxed for applications such as a web server cluster, because the application architecture itself can compensate for the unavailability of individual service nodes.

Note that the third and fourth requirements are not equivalent. It might be possible, for example, to minimize down time by running the upgrade procedure "in the background". But this might conflict with the goal of upgrading the software quickly.

2.3 Solutions to Upgrade/Rollback

We now consider approaches to addressing the challenges of upgrade/rollback. We consider the asynchronous case first, and the synchronous case next.

2.3.1 Asynchronous Upgrade and Rollback

In the asynchronous case, no coordination between nodes is required to effect a successful upgrade. Accordingly, assuming that the application architecture can compensate for the unavailability of individual service nodes, techniques used for upgrade of single nodes can be reused. We briefly discuss two of these techniques.

Packages Given that software is often distributed as packages, and managed by package management systems, such as the Redhat Package Manager, a natural approach is to leverage these systems to manage software updates and rollbacks. Our upgrade procedure might be as follows: shut down the service, copy system configuration files, upgrade the software (taking care to log the prior version of the upgraded software), and then restart the service. The rollback procedure would be to shut down the service, remove the upgraded software, reinstall prior versions, restore configuration files, and then restart the service.

This approach well achieves the first and second goals of Section 2.2. Depending on the number of packages involved in an upgrade, and the complexity of their installation scripts, however, the corresponding rollback operation may take some time to complete. Thus the approach may not achieve the third goal.

File System Checkpoints An alternative approach, that addresses the speed concern (while maintaining the other goals), is to use file system checkpoints. To upgrade the software, we shut down service on a node, checkpoint the filesystem, and then restart service. To roll back the software, we shut down service, roll back the filesystem, and then restart service. Because this approach avoids the need to execute uninstallation and installation code, it may better meet the speed goal.

2.3.2 Synchronous Upgrade and Rollback

We now consider the problem of synchronous upgrades. We consider the specific problem of an upgrade which is both synchronous, and not backwards compatible. An example of such an upgrade would be switching an application's routing protocol from distance vector to link state.

In designing a solution, we draw inspiration from how Internet routing has been upgraded. In moving from IPv4 to IPv6, the Internet has allowed both protocols to operate simultaneously on nodes. Similarly, in our approach, the upgrade process begins with simultaneous execution of old and new versions on the application nodes. The simultaneous execution provides an opportunity to to bootstrap the new application instance, and thus, minimize unavailability due to the upgrade. After the new version is ready to run, we terminate the old instance.

To support simultaneous execution, we employ virtual machines (VMs). VMs provide the illusion of an independent computing machine while running as a process on some other machine. In this context, the VM is referred to as a *guest*, and the other machine is called the *host*.

For our VM, we choose User Mode Linux (UML)[4], which provides a virtual Linux machine running as a process on a Linux host. Two features of UML are required for our purpose. First, it provides the ability to route network packets between the host and guest. Second, it supports copy-on-write filesystem images. To use the copy-on-write facility, the user specifies a base filesystem image, and a copy-on-write file. Both files are stored in the host filesystem. The base filesystem is treated as read-only, and any changes to the guest file system are written to the copy-on-write file.

To employ UML for upgrade/rollback, the distributed application is installed in a UML VM. Before a software upgrade, we duplicate the copy-on-write file and create a snapshot of the running virtual machine process. These files are saved in case a rollback is later required.

Additional copies of the process snapshot and copy-on-write file are then made for the VM that will run the upgraded application. We initialize this VM, using the copied files and the same base filesystem image as the original VM. Next, we perform the software upgrade inside the second VM. Because the two VMs use different copy-on-write files, changes to the filesystem by either VM are not seen in the other VM.

At this point, the system is ready to begin simultaneous execution. If, however, both versions of the application listen on the same network port, we must arbitrate access to that port. With some assistance from the application developer, we can construct viable approaches for both datagram communication (UDP) and byte-stream communication (TCP).

For UDP, we require that the application have a way of identifying and dropping messages from incompatible versions. The application might, for example, include a version number in each message. We then have the host kernel deliver all datagrams for the application to both VMs.

For TCP communication, we cannot simply deliver packets to both VMs. Spurious packets would be processed by the VM's TCP stack, possibly confusing it. Thus the filtering must be done on the host machine. To support this filtering, the application developer provides the host machine with filters that the host machine can use to route application requests to the appropriate application instance. To handle connection establishment, connection requests (SYN packets) are answered by the host machine. After the first request packet from the remote end is received, the application version is identified. The host machine then spoofs a connection request from the remote end to the appropriate VM, discards the VM's response, and forwards the request packet to the VM.

Once the upgrade is complete, we terminate the VM running the older software. How, though, do we determine that an upgrade is complete? Our present approach is to consider the message rate of the old application. As new application nodes enter the system, they choose to run the new application version. Concurrently, old nodes exit the system, decreasing the rate of messages for the old application version. Thus, over time, the message rate for the new version increases, and the message rate for the old version decreases. When the message rate for the old application version at a node drops below a threshold, the node terminates the old application version.

	Process Size (in MB)		
	32	64	128
Snapshot	0.89	1.78	10.42
Resume	< 0.01		

Table 1: Time (in sec) to Snapshot and Resume Processes

In order to ascertain the viability of a VM approach to simultaneous execution, we have conducted some preliminary experiments on the costs of snapshotting and resuming. Because support for resuming UML is not yet complete, we present results based on a process of similar size (in terms of virtual memory image) as a UML VM. Table 1 presents the time to snapshot and resume processes of varying sizes. Experiments were run on a 766 MHz Pentium III with 256 MB of RAM. Snapshot times are short enough that network connections are unlikely to be disrupted. Note that resume times are consistently low, and independent of process size, because resuming only maps the process' data into memory. The data will be faulted in later as needed.

Based on these results, we believe that a virtual machine based approach is appropriate for providing upgrade/rollback facilities for synchronous upgrades. Note that it may be possible to improve the snapshotting time for large processes by performing *lazy snapshots*. For example, we could mark the pages of the virtual machine process as copy-on-write, and then save the snapshot data in the background.

2.4 Open Issues

Open issues remain for both asynchronous and synchronous upgrades. One open question is how to handle side effects of buggy software upgrades. For example, in an e-commerce application, the web service software might be responsible for computing the tax on a purchase. While we can revert a software upgrade that has a bug in its tax computation code, we can not automatically correct its tax calculations. Although such corrections are clearly application specific, incorporating an undo facility [2] may ease the application developer's burden.

Several issues specific to synchronous upgrades remain as well. First, it is not clear that reverting to the prior process state on a rollback is always the right choice. Routing table data, for example, is often transient. Thus, if a large period of time elapses between the upgrade and the initiation of the rollback, it may make more sense to restart the old software with a clean state (an empty routing

table). Also, if the new application version needs to reflect the volatile state of the earlier application, we may need to log and replay messages delivered between the time that the snapshot is taken, and the time that the new software is ready to handle messages.

3 Testing Stage

In the previous section, we described our current efforts for supporting the evolution life-stage of distributed applications. Here, we briefly discuss our future plans for supporting the testing life-stage.

3.1 Test Queries

We envision two basic capabilities to support the testing of distributed applications. One is the ability to inject test queries into the application, and the other is the ability to capture the data that will enable us to verify the correctness and performance of the upgrade. We call the components responsible for these abilities the *injection component* and the *validation component* respectively, and consider the requirements for each component.

The injection component is responsible for delivering test cases into the distributed application for execution. A complication in the implementation of an injection component is that the behavior of a distributed application may depend on who originated a request, or where in the network a request originated. The injection component must reproduce these attributes of the test cases in order to accurately test the deployed distributed application. In particular, this means that the injection component is itself distributed, as test requests may be injected at different nodes of the application.

A possible design for a simple injection component is to require every node to support proxy or relay functionality. This would allow remote developers to masquerade as any node in the system. For example, if a test case requires that a message be sent from node A, then the node running the test simply sends/receives messages to a proxy at node A, which forwards them to the distributed applications (and returns messages from the distributed application to the tester). Messages that need to be sent by a particular user are handled in a similar manner. The ability to impersonate a user or machine naturally raises security issues that must be addressed. In addition, while this allows a developer to route via another node, it may not perfectly recreate the tested node's view of the system. For example, the timing of messages might be affected which in turn may affect the behavior of the system. A richer scripting mechanism to control nodes may provide a more accurate remote view of the system but may be much more complicated to implement.

The validation component is responsible for facilitating the checking of the application on the injected test cases. Note that it does not perform the check itself – the responsibility for determining the correctness of application behavior necessarily requires application specific knowledge. The duty of the validation component is to provide the data that enables validation of application behavior. The data gathering must be lightweight in terms of communications cost. The component must also minimize the effort required by the application developer to specify the data to be captured. At the same time, however, it must provide flexibility in the kind of data to be captured.

An important tool to help validation is support for a rich logging system. A developer should be able to specify a set of triggers, a set of data types, and a callback for each data type. Incoming messages at a node can be matched against the set of triggers and the callbacks are invoked to capture data when a message matches a trigger. The triggers can be checked before or after the message is processed by the application. Some key open challenges are the scalable distribution of these triggers and callbacks to the application nodes and the scalable collection of the resulting logs for analysis. It is likely that not all nodes in the system will be involved in the validation, thereby simplifying the task of distribution and collection.

3.2 Debugging Support

When developers of traditional applications need to understand application behavior, they often employ source-level debuggers. A debugger provides a developer the ability to study program behavior at a microscopic level. Specific abilities of common debuggers include single-stepping through program execution, live inspection of complete program state, including memory and registers, and post-mortem analysis of program state via *core* files.

How might we provide similar capabilities to developers of networked applications? As a first, naïve, approach, consider building a debugging system for distributed applications using traditional debuggers and a remote invocation facility such as *ssh*. In this approach, the remote invocation facility is used to attach a debugger process to each process of the dis-

tributed application. Unfortunately, this approach approach does not work well. To illustrate, we consider how such a system deals with single-stepping.

How do we implement single-stepping in this system? This is, perhaps, a trick question. While single-stepping is well-defined for a single process, there is no clear analogue for an application with multiple processes. There is *no* obvious order in which we could instruct the processes to execute their next steps, because concurrent execution is rife with the potential for race conditions. Hence, it does not matter which implementation we choose: single-stepping is *inherently* an inappropriate debugging primitive for distributed applications. One important challenge, then, is to find a set of primitives that provide meaningful and useful semantics for distributed applications.

One possibility is to support a less demanding form of application debugging. For example, instead of using traditional debuggers at each node, message logging may provide a useful first approximation to single stepping. Clearly, capturing all the messages generated at all nodes could be very expensive for some applications. Instead, we may limit the log to those messages associated with certain requests, restricted either by message type, or by node using the techniques described for validation in Section 3.1. Sahai et al. [8] propose a method of capturing all the messages associated with a request that may be useful for this approach. Either synchronized clocks or Lamport clocks could be used to timestamp events in these logs. Developers could perform real-time visualization of the logs to observe their application in operation. Alternatively, if all messages are collected it might be possible to diagnose problems by replaying the logs within a simulator, where traditional debugging would be possible.

4 Conclusions

It is clear that distributed applications are a significant part of the computing landscape. But to fully realize the potential of networked computing, we will need richer development environments that lessen the burden on application developers. We believe that the development of distributed applications can be improved by an appropriate life-cycle model, and tools that support that model.

We have proposed a life-cycle that consists of design, implementation, testing, deployment and operation, and maintenance and evolution stages. Tools to support the first two stages already exist, or are now emerging. For the maintenance and evolution phase, initial work indicates that approaches based on versioning/snapshot filesystems and virtual machines will be appropriate for many distributed applications. To support the testing phase, we propose the message injection and data capture primitives. We also find that traditional approaches to debugging do not map well to distributed applications, and that new primitives will be needed. As a start, we propose rich forms of message logging as an approximation of single-stepping. Finally, while we have not addressed the issue of the remaining life-stages in this paper, we hope to eventually develop a tool chain that helps developers in all stages.

References

[1] Iris: Infrastructure for resilient internet systems. http://iris.lct.mit.edu/.

[2] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX 2003 Annual Technical Conference*. San Antonio, TX, Jun. 2003.

[3] Y.-H. Chu, S. G. Rao, et al. A case for end system multicast. In *Proceedings of the ACM Sigmetrics 2000*, pp. 1–12. ACM, Santa Clara, CA, Jun. 2000.

[4] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the Annual Linux Showcase*. Atlanta, GA, Oct. 2000.

[5] B. Liskov. Software upgrades in distributed systems. Invited Talk, 18th ACM Symposium on Operating Systems Principles, Oct. 2001.

[6] ns-2 Network Simulator. http://www.isi.edu/nsnam/ns/, 2000.

[7] L. Peterson, T. Anderson, et al. A blueprint for introducing disruptive technology into the internet. In *Proceedings of ACM HotNets-I Workshop*. Princeton, New Jersey, Oct. 2002.

[8] A. Sahai, V. Machiraju, et al. Message tracking in SOAP-based web services. Tech. Rep. HPL-2001-199, HP Labs, 2001.

[9] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th Symposium on Operating System Principles*. Dec. 1999.

[10] B. White, J. Lepreau, et al. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 255–270. USENIX Association, Boston, MA, Dec. 2002.

Virtual Appliances in the Collective: A Road to Hassle-Free Computing

Constantine Sapuntzakis and Monica S. Lam
Computer Systems Laboratory
Stanford University

Abstract

This paper describes the vision of the Collective, a compute utility which runs internet services as well as the highly interactive applications we run on desktop computers today. As part of this vision, we wish to shift the burden of administering the desktops from users to professionals. To decrease the cost of administering systems, we find inspiration in the reliability and maintainability of network-connected computer appliances. We argue for structuring our software as a group of networked appliances, each appliance virtualized on a virtual machine monitor. We show how to run virtual appliances in the Collective system and examine some ways in which individuals and groups may adopt virtual appliances.

1 Introduction

The progress of computer hardware has given us abundant computation, storage, and communication capacity. Our only limitation seems to be our ability to use and manage this wealth. The challenge is to develop software that makes compute resources into a utility as easy to use as water, power, and the telephone.

Users expect the following from a compute utility:

1. *Global, uniform access.* Users should have access to their computing environment anywhere in the world, as if they were in their own offices. The computing environment should include not just web services but all the applications people run on their computers today.

2. *Hassle-free computing.* Users today spend too much time administering their machines, that is, if they know how. After all, many home computer users do not backup their systems or apply the latest security patches. In a utility, computer administration must disappear into the infrastructure, becoming invisible to users.

3. *An open system.* To encourage competition in delivering robust, easy-to-use software systems, a utility must not require that applications be run on a particular operating system or be written in a specific programming language.

4. *Security in a public infrastructure.* A user would like the computers in the utility to perform computations correctly on their behalf, to respect their privacy, and to not corrupt their data. We foresee that a combination of laws, such as those against interfering with the mail, and technologies, like frequent backups, trusted computing, intrusion detection, and auditing, will give users enough confidence to move their computation to a utility. Trust is a matter of extent. Just like with the credit card system, users will not expect the utility to be completely trustworthy. A company may choose to operate on its sensitive data only on the company's internal compute utility.

Today's computing environments are a far cry from the compute utility described above. This paper presents the high-level design of our Collective system architecture, an attempt to create this utility. As a first step, we look at ways to make computing environments more reliable and easier to manage. Inspired by computer appliances, which trade off generality for ease of use and reliability, we propose to structure our computing environments as collections of appliances. By virtualizing the appliances, multiple appliances can run on one piece of hardware, making the model affordable for a far wider range of software and users.

This paper describes how the ideas in the Collective can change our computing landscape. It makes the following main points. First, we describe the vision of a future computing environment where compute services are regarded as a utility. Second, we argue for splitting a computer's software into multiple virtual appliances. Third, we briefly describe the functions of the Collective system software. Fourth,

we discuss the socio-economic ramifications of this proposed architecture. Although the Collective is designed with the goal of creating a global utility, certain ideas, like virtual appliances, can be readily adopted to solve today's software management problems.

2 Design Concept

Because PCs are hard to use and manage, some have predicted that appliances will become popular [13]. This section describes how we can borrow ideas from the design of appliances to improve the manageability and usability of computers.

2.1 Appliances

With the falling cost of computer hardware, special-purpose appliances abound, e.g. firewalls, VPN gateways, game consoles, TiVos, and NetApp filers [7]. While some of these appliances are built out of PC hardware and run PC operating systems, appliances differ from PCs in several ways. For one, an appliance does not try to do everything and, as a result, is easier to use and maintain. The appliance comes with the software needed to serve its purpose. The appliance maker tests all the software to ensure it works together; on the PC, the user is often mating an application with a version of the operating system or libraries that it was never tested with. Finally, two appliances are better isolated than two application installed on the same operating system, reducing the chance that the bad behavior of one will harm the other.

There is one more, perhaps most significant, difference: appliances, especially networked ones, are maintained by the makers, not the users. On an appliance, the maker controls all the software and can create correct updates with higher confidence. Network-connected appliances such as the TiVo download updates periodically to fix bugs, add new features, and plug security holes. In contrast, on Windows and Linux, users must initiate software updates; makers are nervous about breaking a user's configuration.

2.2 Using Appliances to Structure the Computing Environment

To gain ease of use and manageability, we can structure our computing environments as groups of appliances. A user might have an appliance for each application he uses today, for example, an AOL appliance, an office suite appliance, and a video editing appliance. A user may even have multiple appliances with similar software: an office suite appliance for work, and an office suite appliance for personal correspondence. A user may wish to bundle multiple appliances into a single unit. For example, a company might want to make sure that a telecommuter's office suite appliance is protected by the company's firewall/VPN appliance and audited for break-ins by an intrusion detection appliance.

By placing proxy appliances at the network ports of a current appliance, we can roll out new network protocols without modifying currently running appliances. IPsec can be deployed in an encryption/authentication appliance. A new network file system can be deployed with a translator to and from NFS. The proxies can be implemented at user level as packet filters without worrying about deadlocks.

Each appliance is connected to the network and maintained by the maker. The user extends their environment by getting more appliances. Network protocols can be used to promote sharing between appliances, like network cut-and-paste[11] and shared network file systems for user files.

Still, hardware appliances have their limitations. Hardware is expensive realtive to software, takes space, power, generates noise and heat, and must be physically delivered. An appliance's hardware can fail, potentially trapping configuration and user state, making it hard to recover.

2.3 Virtual Appliances

Many of the limits described in the previous section can be overcome by making appliances virtual. A virtual appliance is the state of a real appliance (the contents of the appliance's disks) as well as a description of the hardware (e.g. two Ethernet adapters, 256mb RAM, two hard disks, etc.). Since a virtual appliance is just data, it can be shipped electronically. A virtual appliance runs in a virtual machine monitor (VMM), sits on a virtual network, and stores data on virtual disks or network storage. The virtual appliance talks over the network to real I/O devices, like displays, printers, game pads, and keyboards.

Using a virtual machine monitor (VMM), like VMware GSX server[16], we can run many virtual appliances on a single computer, spreading the cost, power, space, and heat over multiple appliances. This will make the appliance model affordable for a wider range of applications.

We can run the same software that was on the hardware appliance. Since the VMM hides differences in the physical hardware, the appliance maker can maintain a small set of device drivers and still have its appliances run on wide range of hardware. To ease the

transition to a networked world, the VMM can map the virtual appliance's hardware devices and protocols to network devices and protocols. While a virtual appliance may think that its computer has a hardware display adapter, the virtual display will actually talk over the network to a remote display on a thin client. A virtual appliance may think that it is talking to a local hard disk, but instead the hard disk is hosted on a reliable network storage service.

3 Collective: A Network of Hosts of Virtual Appliances

In this section, we describe the Collective architecture. The Collective is a compute utility based on the concept of appliances. In the Collective system, machines serve as caches of virtual appliances, and the states of appliances are saved in some persistent data store.

The Collective software uses the VMware x86 virtual machine to execute, resume, and suspend virtual appliances. There are many advantages to virtualizing the x86 architecture. The machine can run any virtual appliance that runs on x86 hardware, a de facto standard. It does not require the software to run on any particular operating system. Because the operating system is included in the appliance, system administration is performed by the appliance makers, not the users. Finally, because the VMware virtual machine monitor is a commercial product, we can run experiments on a usable prototype system.

The computers that run virtual appliances are called hosts. To create a utility out of these hosts, the Collective software provides the following additional functions:

1. *Virtual networks of virtual appliances.* To implement a network of virtual appliances, not only does the Collective provide a virtual machine interface, it also virtualizes the network. It uses (1) Ethernet virtual LANs (or VLANs) to connect virtual appliances on separate physical hosts, and (2) virtual Ethernet switches on the same machine to create multiple isolated networks within a single host.

2. *A networked service plane.* The Collective provides a "service plane" that automates the management of virtual appliances and hardware resources. The Collective keeps the virtual appliances up to date, replicates them, migrates them as needed to present the user with the illusion that he has instantaneous and fast access to all the latest appliances wherever he goes. This is a challenge since the state of an x86 machine can be large. Our previous work proposed various optimizations to enable new appliances to start up quickly, to reduce the amount of traffic needed to update an appliance, and to speed up the migration of appliance states between machines[14]. The service plane will also perform optimizations such as load balancing to increase the utilization of hosts in the system.

3. *Introspective facilities.* Having access to the state in appliances and running as a separate entity, the Collective can provide introspective services to add features to appliances. For example, the Collective can examine the state of an executing appliance to detect signs of intrusion[4]. This is superior to implementing intrusion detection in the appliance because the detector itself would have to guard against being compromised. Another example is a general checkpointing facility for error recovery. The Collective can checkpoint the state of an appliance as it executes so that users can access a prior appliance state should an error occur.

4. *A trusted computing platform.* Before we run a job on a machine in the utility, how can we tell that the software on the host is not malicious and will respect our security and privacy concerns? One option is to ensure the host is running a trusted virtual machine monitor[5], attested to by tamper-proof hardware.

The above sketches our high-level approaches towards providing the properties desired of a utility, as described in Section 1. The service plane of the system migrates appliances efficiently to give users global access to their computing environment. The concept of actively managed virtual appliances reduces the hassles in computing. Openness is achieved by adopting the x86 architecture interface. Finally, ideas like TCPA are used to provide some degree of trust in the infrastructure.

4 The Socio-Economic Landscape

Creating a global utility is an ambitious goal. Not only are there many technical details to work out, it is important that economic incentives be in place to make such a system happen. It is important that we can stage the development by creating subsystems that address some of the real problems we face today. In this way, we can gain valuable experience needed

to build the ultimate system. In the following, we describe how a subset of the ideas described above can be used to solve real problems encountered by today's computer users both at homes and in the workplace.

4.1 Information Technology for Organizations

Manpower, not hardware cost, dominates the information technology (IT) spending in many large organizations. IT departments provision one system administrator for every 20 to 50 computers.

The Collective will make it easier to deploy and maintain turnkey solutions for many markets, decreasing the IT staff, and perhaps eliminating it in many small and medium-sized organizations. Organizations which benefit from unique IT processes or require special applications will continue to retain an IT staff, much like they retain one today to deploy applications and services. For example, an IT department may make a special bundle of appliances for finance which is different from the bundle for human resources. Professors may have a set of appliances for their research and another set for sharing with students.

The Collective can also ease these IT tasks:

1. *Unauthorized applications.* The IT department can provide a set of working, core appliances firewalled from the rest of a user's setup. Even if a user adds other, unauthorized or untested appliances, the isolation and firewalling of the appliance model should keep the core working, even though the new appliances may break or misbehave. If, after experimenting with the appliance in isolation, the appliance turns out to be useful, the IT department can give it more access to other core appliances and data or even make it part of the core.

2. *Setting up new offices.* We can set up a new office quickly by simply buying the additional hardware and connecting it to the network. Complete appliances from the organization's headquarters or purchased from third-party companies will automatically populate the machines as employees use their appliances.

3. *Administration of branch offices.* Some organizations have many branch and sales offices dispersed geographically. With our architecture, there is no difference between the employee's experience in the headquarters or the branch offices, because the relevant appliances in all the offices are updated automatically.

4. *Relocation and telecommuting.* With our architecture, a user can access their running appliances from any machine. The system will automatically migrate and cache the appliances to give users fast response. With this feature, employees can work at home or move between offices without worrying about moving files or restarting applications.

5. *Error and disaster recovery.* All of the appliances can be backed up at a remote location and retrieved if errors are discovered or disaster strikes. In addition, our system's introspective ability can be used to save the active state of an appliance as it executes. This is useful for recovering from errors not just in the software and hardware, but also from operator errors.

4.2 Home Users

It is ironic that professionals in enterprises and universities are supported by system administrators, whereas novice home users are not. How do we expect novice users to know about backups, apply security patches, and run virus detectors and disk defragmenters? We believe that plenty of frustrated home users will gladly move to an easy-to-use and maintenance-free collection of virtual appliances.

Instead of buying a PC, consumers would buy a "universal appliance host" which bundles the x86 processor with a thin layer of software, including a VMM. User files will be managed by a storage service, which keeps the user's data locally and as well as encrypted backups at a remote site.

We expect that there will be many companies who specialize in developing attractive easy-to-use virtual appliances for each market segment. (This model does not preclude the development of free and personal appliances.) There will be appliances for senior citizens, novice users, hobbyists of different kinds, teenage girls, teenage boys, and children of different ages. Each of these appliances will combine a large number of software titles, some of which may have been developed by a third party. There will be more titles than any individual would typically install on his machine. Or, if the features in a single appliance do not suffice, people may run multiple appliances.

Users will rent or subscribe to the appliances; using individual software titles in the appliance may have additional fees. In return for paying a fee, appliances will be actively maintained and updated by these companies. The predictable update model provided by the Collective will keep the costs of the service low.

4.3 Large-Scale Services

It is interesting to compare the proposed software model with services provided by portals such as AOL and Yahoo. These portals give their customers services such as email, browsing and instant messaging. These services are kept up-to-date; users get the benefit of new features such as spam filters, virus scanners, and parental control, without having to modify their own machines. And, users can get access to these services anywhere they go.

Portal computing has its disadvantages. To serve a large number of clients, portals offer services that are not too computationally demanding. With a global utility like the Collective, there can be a continuum between central portal services and distributed user appliances. Using the Collective, the central service can replicate itself to handle load or improve interactivity. Depending on how state is shared, the service can choose to partition state across the replicas. Desktop appliances are the extreme. Since there is little sharing, the service is partitioned to handle only one user's session; the service is placed at the user's computer for good interactivity.

Today, data is trapped at the portal, making it harder to use across portals. For the convenience of not having to manage software, many users have chosen to entrust their private data to unknown companies that may go under, selling or destroying their data. In the Collective, users will bring their own storage to the service.

5 Related Work

Sun's N1[15], IBM's "autonomic computing"[1], Duke's "Cluster-on-Demand"[12], and HP "utility data center"[8] all aim to simplify the mapping of services onto pools of computers, networks, and storage. Grid computing aims to provide a large pool of computing for scientific applications[3]. In the Collective, we aim to manage not just web services and scientific applications but highly interactive applications that reside today on the desktop.

Goldberg surveyed the field of virtual machines[6]. More recently, Disco revitalized interests in virtual machines; a major change from the previous work had been the development of computer networks and network protocols to share data easily and quickly between multiple computers[2]. Internet suspend/resume describes how to use the VMware VMM to provide user mobility in the wide area[10]. The Collective uses the VMware VMM in the same fashion to provide user mobility[14]. The Denali isolation kernel shows that with a couple of minor architectural modifications, it is possible to scale up to hundreds of virtual machines on a single computer[17].

Java provides mobile code in a portable virtual machine but requires users to rewrite their code in Java and to new interfaces[9]. In contrast, virtualizing using a VMM allows us to use the large amounts of code already written to run on today's hardware platforms, including Java.

6 Summary

By structuring software as a group of network-connected appliances, users will manage software less. With appliances, the maker controls the software installed on the appliance, allowing the maker to update the software without user intervention and with predictable results. Users can still extend their environments by adding appliances. Virtualizing appliances makes them cheaper and more manageable. This allows us to apply the appliance concept to more users and more applications.

While an end user or small business will likely subscribe to appliances that suit their needs and tastes, large companies will continue to have an in-house IT department, which will use appliances to manage the company's computing assets.

A Collective-like global utility enables software to be replicated, partitioned, and pushed into the network by encapsulating it in appliances, forming a continuum between central portal services and distributed desktop applications.

7 Acknowledgements

This work was funded in part by the National Science Foundation under Grant No. 0121481 and Stanford Graduate Fellowships. We thank Ramesh Chandra and Mendel Rosenblum for discussions on the ideas of this paper.

References

[1] Autonomic computing. `http://www.research.ibm.com/autonomic/`.

[2] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412--447, November 1997.

[3] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37-46, 2002.

[4] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Internet Society's 2003 Symposium on Network and Distributed Systems Security*, February 2003.

[5] T. Garfinkel, M. Rosenblum, and D. Boneh. A broader vision of trusted computing. In *Proceedings of the Ninth Workshop on Hot Topics in Operating System*, May 2003.

[6] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[7] D. Hitz. A storage networking appliance. Technical Report TR3001, Network Appliance, Inc., October 2000.

[8] HP utility data center. `http://www.hp.com/solutions1/infrastructure/solutions/utilitydata/`.

[9] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[10] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 40-46, June 2002.

[11] R. Miller and B. Myers. Synchronizing clipboards of multiple computers. In *Proceedings of the Twelfth Symposium on User Interface Software and Technology*, pages 65-66, November 1999.

[12] J. Moore and J. Chase. Cluster on demand. Technical report, Duke University, May 2002.

[13] D. Norman. *The Invisible Computer*. MIT Press, 1998.

[14] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 377–390, December 2002.

[15] Sun N1. `http://wwws.sun.com/software/solutions/n1/index.html`.

[16] "GSX server", white paper. `http://www.vmware.com/pdf/gsx_whitepaper.pdf`.

[17] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, pages 195–210, December 2002.

POST: A secure, resilient, cooperative messaging system*

Alan Mislove[1] Ansley Post[1] Charles Reis[1] Paul Willmann[1] Peter Druschel[1]
Dan S. Wallach[1] Xavier Bonnaire[2] Pierre Sens[2] Jean-Michel Busca[2]
Luciana Arantes-Bezerra[2]

[1]*Rice University, Houston, TX, USA*
[2]*LIP6, Université Paris VI, Paris, France*

Abstract

POST is a decentralized messaging infrastructure that supports a wide range of collaborative applications, including electronic mail, instant messaging, chat, news, shared calendars and whiteboards. POST is highly resilient, secure, scalable and does not rely on dedicated servers. Instead, POST is built upon a peer-to-peer (p2p) overlay network, consisting of participants' desktop computers. POST offers three simple and general services: (i) secure, single-copy message storage, (ii) metadata based on single-writer logs, and (iii) event notification. We sketch POST's basic messaging infrastructure and show how POST can be used to construct a cooperative, secure email service called ePOST.

1 Introduction

Messaging systems like traditional email and news, as well as instant messaging, shared calendars and bulletin boards, are among the most successful and widely used distributed applications. Today, these services are implemented in the client-server model. Messages are stored on and routed through dedicated servers, each hosting a set of user accounts. This partial centralization limits availability, because a failure or attack on a server denies service to the users it supports. Also, substantial infrastructure, maintenance and administration costs are required to scale to large numbers of users. This is true in particular for semantically rich, complex messaging systems like Microsoft Exchange and Lotus Notes.

POST is a cooperative infrastructure that utilizes the untapped resources of users' desktops to provide messaging services. Unlike server-based systems, POST is *self-scaling*: the addition of new user desktops and periodic upgrades of existing desktops implicitly add more resources, thus balancing increased demands on the service due to additional users and new features. POST does not present a single point of failure or attack, and is thus potentially more resilient than server-based systems. Finally, the self-organizing properties of POST promise reduced system administration costs.

POST provides three basic services to applications: (1) persistent single-copy message storage, (2) metadata based on single-writer logs, and (3) event notification. A wide range of messaging applications can be constructed on top of POST using these services.

POST is built upon a structured p2p overlay network, providing it scalability, resilience and self-organization. Users contribute resources to the POST system (CPU, disk space, network bandwidth), and in return, they are able to utilize its services. POST assumes that participating nodes can suffer byzantine failures. Stronger failure assumptions may be unrealistic, even in scenarios where participating hosts belong to a single organization, because a single compromised node could disrupt critical messaging services or disclose confidential messages.

In this paper, we sketch the design of POST, and then describe how a cooperative, secure email system can be built using POST. Unlike conventional email services, our *ePOST* system provides secure email services by default and requires no dedicated servers. Furthermore, due to its strong sender authentication, ePOST makes efficient spam defense easier. We chose email as the initial application for POST because it is well understood, and because its high availability, reliability and security demands make it a challenging driver for POST and p2p systems in general.

The remainder of this paper is organized as follows. Section 2 provides background information on Pastry, PAST, and Scribe. Section 3 sketches the design of the POST infrastructure. In Section 4, we sketch the design of a cooperative email system as an example POST application. Section 5 discusses integrating ePOST with existing email systems. Section 6 outlines related work, and Section 7 concludes.

2 Background

POST relies on *Pastry*, a structured overlay network, as well as two basic services built upon Pastry: *PAST*, a

*This research was supported in part by Texas ATP (003604-0079-2001) and by NSF (ANI-0225660). http://project-iris.net.

storage system and *Scribe*, a group communication system. POST could easily be layered on similar systems like Chord/CFS, or Tapestry/OceanStore [14, 6, 9, 16].

Pastry [12] is a structured p2p overlay network designed to be self-organizing, highly scalable, and fault tolerant. In Pastry, every node and every object is assigned a unique identifier chosen from a large id space, referred to as a *nodeId* and *key*, respectively. Given a message and a key, Pastry can efficiently route the message to the node whose nodeId is numerically closest to the key.

PAST [13] is a storage system built on top of Pastry and can be viewed as a distributed hash table. Each stored item in PAST is given a 160 bit key (hereafter referred to as the *handle*), and replicas of an object are stored at the k nodes whose nodeIds are the numerically closest to the object's handle. PAST maintains this invariant regardless of node arrivals or failures. Since nodeId assignment is random, these k nodes are unlikely to suffer correlated failures. PAST relies on Pastry's secure routing [2] to ensure that k replicas are stored on the correct nodes, despite the presence of malicious nodes. Throughout this paper, we assume that at most $k-1$ nodes are faulty or unreachable in any replica set.

POST stores three types of data in PAST: *content-hash blocks*, *public-key blocks*, and *certificate blocks*. Content-hash blocks are stored using the cryptographic hash of the block's contents as the handle. Public-key blocks contain monotonically increasing timestamps, are signed with a private key, and are stored using the cryptographic hash of the corresponding public key as the handle. Certificate blocks are signed by a trusted third party and bind a public key to a name (e.g., an email address). The block is stored using the cryptographic hash of the name as the handle.

Content-hash blocks can be authenticated by obtaining a single replica and verifying that its contents match the handle. Unlike content-hash blocks, public key blocks are mutable. To prevent rollback attacks by malicious storage nodes, clients attempt to obtain all k replicas and choose the authentic block with the most recent timestamp. Certificate blocks require a signature verification using the public key of a trusted third party.

Scribe [3] is a scalable multicast system built on top of Pastry. Each Scribe group has a 160 bit *groupId*, which serves as the address of the group. The nodes subscribed to each group form a multicast tree, consisting of the union of Pastry routes from all group members to the node with nodeId numerically closest to the groupId.

3 POST Architecture

POST provides three basic services: a shared, secure single-copy message store, metadata based on single-writer logs, and event notification. These services can be combined to implement a variety of collaborative applications, like email, news, instant messaging, shared calendars and whiteboards.

A typical pattern is that users create messages and insert them in encrypted form into the secure store. To send a message to another user or group, the notification service is used to provide the recipient(s) with the necessary information to locate and decrypt the message. The recipients may then modify their personal metadata to incorporate the message into their view (e.g., into a private mail folder).

POST assumes the existence of a certificate authority. This authority signs certificates binding a user's unique name (e.g., her email address) to her public key. The same authority issues the nodeId certificates required for secure routing in Pastry [2]. Furthermore, the authority may set policies for each user (such as ensuring that each user owns a nodeId bound to a live IP address), thus forcing the user to contribute resources to the system. Users can access the system from any node, but it is assumed that the user trusts her local node, hereafter referred to as the trusted node, with her private key.

Throughout the design of POST, we assume that objects stored in PAST cannot be deleted. Thus, the amount of available disk space in the system must be increasing and greater than the total storage requirements, which is reasonable to expect in a p2p environment where each participant is required to contribute a portion of her desktop's local disk.

3.1 User Accounts

Each user in the POST system possesses an account, which is associated with an identity certificate. The certificate is stored as a certificate block, using the secure hash of the user's name as the handle. Also associated with each account is a user identity block, which contains a description of the user, the contact address of the user's current trusted node, and any references to public metadata associated with the account. The identity block is stored as a public-key block, signed with the user's private key. Finally, each account has an associated Scribe group used for notification, with a groupId equal to the cryptographic hash of the user's public key.

The immutable identity certificate, combined with the mutable public-key block, provides a secure means for a trusted authority to bind names to keys, while giving users the ability to change their personal contact data without requiring subsequent interactions with the certificate authority. The Scribe group allows anybody waiting for news from that user, or anybody wishing to notify the user that new data is available, to have a common rendezvous point.

3.2 Secure Message Storage

POST provides a shared, secure message storage facility. Application-provided message data is encrypted using a technique known as convergent encryption [7]. Convergent encryption allows a message to be disclosed to selected recipients, while ensuring that copies of a given cleartext message inserted by different users map to the

same ciphertext, thus requiring only a single copy of the ciphertext to be stored.

When an application wishes to store message X, POST first computes the cryptographic $Hash(X)$, uses this hash as a key to encrypt X with an symmetric cipher, and then stores the resulting ciphertext at the handle

$$Hash\left(Encrypt_{Hash(X)}(X)\right)$$

which is the secure hash of the ciphertext. To decrypt the message, a user must know the hash of the plaintext.

Convergent encryption reduces the storage requirements when multiple copies of the same content tend to be inserted into the store independently. This happens commonly in cooperative applications, for instance, when a given popular document is sent as an email attachment or posted on bulletin boards by different users.

Convergent encryption is vulnerable to certain known plaintext attacks. An attacker who is able to guess the plaintext of a message can verify its existence in the store, but cannot necessarily find out who inserted it. Moreover, since users normally encrypt their private metadata, it is not possible to determine who references the message. Nevertheless, with sensitive content, convergent encryption must be used with care, particularly when the content is of a small size, is highly structured, or is otherwise predictable. In such cases, convergent encryption could be supplemented or replaced by conventional cryptographic methods. A simple change could be to prepend some number of random bits to the plaintext prior to the convergent encryption.

3.2.1 Scoped storage overlays

P2p storage systems like PAST or CFS form a single overlay network that includes all participants. Replicas of stored objects are placed at random nodes with adjacent nodeIds throughout this overlay. This approach leads to good load balancing and failure independence, since the set of replica nodes for an object is widely distributed and thus unlikely to suffer correlated failures.

On the other hand, network locality can be poor because all objects are replicated at global scope, even when an object is only of local interest and a more local distribution (e.g., within a large organization) may yield adequate failure independence. The lack of centralized node administration makes it difficult to assess individual nodes' failure probabilities, and thus determine the appropriate degree of replication. And, the fact that any node can insert objects anywhere in the system invites denial-of-service attacks aimed at exhausting the storage space of certain nodes, or the entire system. Lastly, it is difficult to let nodes behind a firewall participate in the storage overlay.

POST overcomes these problems using a two-level store consisting of organizational overlays and a global overlay. The two-level store allows POST to scope the insertion of documents into the store, such that documents inserted by members of an organization are replicated among the organization's nodes. This is achieved without sacrificing load balancing, failure independence, or the ability to look up a stored message anywhere in the global overlay; we omit the details due to lack of space.

3.3 Event notification

The event notification service is used to alert users to certain events, such as the availability of a message, a change in the state of a user, or a change in the state of a shared object.

For instance, after a new message was inserted into POST as part of an email or news service, the intended receiver(s) must be alerted to the availability of the message and provided with the appropriate decryption key. Commonly, this type of notification requires obtaining the contact address from the recipient's identity block. (This may require a lookup of the recipient's certificate block, if the certificate is not already cached by the sender). Then, a notification message is sent to the recipient's contact address, containing the secure hash of the message's ciphertext and its decryption key, encrypted with the recipient's public key and signed by the sender.

In practice, notification can be more complicated if the sender and the recipient are not on-line at the same time. To handle this case, the sender may delegate the responsibility of delivering the notification message to a set of k random nodes; we omit the details here due to lack of space.

To guarantee confidentiality, each notification message is encrypted using a symmetric cipher such as AES with a unique session key, and the session key itself is then encrypted using the recipient's public key. Thus, only the recipient can decrypt the session key (i.e., with his private key) in order to decrypt the remainder of the message. Each notification message is also signed with the sender's private key, allowing the recipient to verify its authenticity. Finally, each notification message also includes a timestamp to prevent the message from being replayed by malicious users. Note that, unlike most traditional user messaging infrastructures, everything in POST is digitally signed and encrypted, by default. This will prove useful when implementing higher-level services like email, chat, and so forth.

3.4 Metadata

POST provides single-writer logs that allow applications to maintain metadata. Typically, a log encodes a view of a specific user or group of users and refers to stored messages. For instance, a log may represent updates to a user's private email folder, or a public news group. An email or news application would use a log of insert and delete records to keep track of the state of a user's mail folder or a shared folder representing a news group.

In general, logs can be used to track the state of a chatroom, a newsgroup, a shared calendar, or an arbitrary data structure. POST represents logs using self-authenticating blocks that form a content-hash chain. This is similar to, and was inspired by, the logs used in the Ivy p2p filesystem [11].

The log head is stored as a public-key block and contains the location of the most recent log record. Handles for log heads may be stored in the user's identity block, in a log record, or in a message. Each log record is stored as a content-hash block and contains application-specific metadata and the handle of the next recent record in the log. Applications optionally encrypt the contents of log records depending on the intended set of readers.

In the original implementation used in Ivy, the log head and each log record are stored at a different set of nodes. To allow for more efficient log traversal, POST stores clusters of M consecutive log records on the same node, under the handle of the least recent of the M records. To deal with partially filled clusters, the log head contains an additional handle, referring to the least recent record in a partially filled cluster. This handle identifies the cluster.

Other optimizations are possible to reduce the overhead of log traversals, including caching of log records at clients and the use of snapshots. Like Ivy, POST applications may periodically insert snapshots of their metadata into the store. Thus, log traversals always terminate at the most recent snapshot.

3.5 POST robustness and security

The single-writer property and the content-hash chaining [10] of the logs make it very hard for a malicious user or storage node to insert a new log record or to modify an existing log record without the change being detected. To prevent version rollback attacks, public-key blocks contain version timestamps. When reading a public-key block (e.g., a loghead) from the store, clients attempt to read all k replicas of the block, and use the authentic replica with the most recent timestamp. When reading content-hash blocks or certificate blocks, it is sufficient to use any authentic replica.

Of great concern is the durability of stored messages. It depends on the failure independence of the replica node sets and an appropriate choice of replication factor, relative to the failure rate of individual nodes. POST's scoped insertion into local overlays greatly eases the assessment of failure independence and node failure rates, because all nodes are under some level of joint administrative control.

Organizations that run a local overlay should ensure that nodes are spread over different buildings, if not different sites. To reduce the risk of correlated failures due to security attacks, there should be sufficient heterogeneity in hardware and software. This can be difficult to ensure due to most organizations' monoculture approach to systems administration. However, risks from common virus attacks can be greatly reduced by running the POST daemon with reduced system privileges under its own user identifier. Thus, a compromised POST daemon has insufficient privileges to cause harm to the rest of the system. Likewise, other compromised user applications cannot attack POST's local file store.

Pastry's secure routing mechanism provides an effective defense against denial-of-service attacks against the overlay, both from within and outside [2]. Attacks aimed at filling the store can be thwarted with relative ease due to the use of local overlays. Since object insertions are allowed only within a local overlay, it is possible to track, identify and reprimand offenders within an organization.

Single-writer logs are the only mechanism used to maintain mutable state in POST. Their use avoids the cost and complexity of a general byzantine fault-tolerant replicated state machine. We are confident that POST's restricted mechanism for mutable state is flexible enough for applications like email, news, instant messaging and calendaring. The logs are efficient in cooperative applications, where insertions occur at a rate typical of human user actions.

Some cooperative applications may require a more flexible mechanism for maintaining mutable state. To support such applications, the authors at LIP6 are currently investigating additional, byzantine fault-tolerant mechanisms for maintaining multi-writer, mutable state.

4 Example: Electronic mail

In this section, we sketch the design of a serverless email system, ePOST, on top of the POST infrastructure. The goal is to show how POST can support a secure, scalable and highly resilient email system that leverages the resources of participating desktop computers.

While a system like ePOST promises increased resilience, greater scalability and lower cost, it remains an open question whether these advantages will be sufficient to completely displace the existing, server-based email infrastructure. Nevertheless, we chose to pursue ePOST for several reasons.

First, ePOST is designed so that it can be deployed incrementally, thus allowing individual organizations to adopt it while still relying on existing standards and infrastructure for communication across organizations. Second, unlike most existing p2p applications, email is mission-critical and demands high reliability, security, and availability. Thus, it is a challenging driver for the development of POST and, more generally, the underlying p2p infrastructure.

4.1 Overview

Each ePOST user is expected to run a daemon program on his desktop computer that implements the Pastry, PAST, Scribe and POST protocols, and contributes some CPU, network bandwidth and disk storage to the system. The daemon acts as a SMTP and IMAP server, thus allowing the user to utilize conventional email client programs. The daemon is assumed to be trusted by the user and holds the user's private key. No other nodes in the system are assumed to be trusted by the user (other than the authority that signs the users' certificates).

4.2 Email storage

In ePOST, email messages received from an email client program are parsed and the MIME components of the message (message body and any attachments) are stored as separate messages in POST. Thus, frequently circulated attachments are stored in the system only once.

The message components are first inserted into POST by the sender's ePOST daemon; then, a notification message is sent to the recipient. Sending a message or attachment to a large number of recipients requires very little additional storage overhead beyond sending to a single recipient. If messages are forwarded or sent by different users, the original message data does not need to be stored again; the original message reference is reused.

Due to the necessary data replication in PAST, the storage overhead per message is higher in POST compared to a conventional server-based email system. However, this effect is partly offset by POST's single-copy store, which eliminates large amounts of duplication due to large, widely circulated email attachments. Moreover, exploiting the typically underutilized disk space on desktop computers should more than compensate for this overhead [1]. Lastly, the storage requirements can be further reduced by using erasure codes [15], but we have not yet explored their use in POST.

4.3 Email delivery

The delivery of new email is accomplished using POST's notification service. A sender first constructs a notification message containing basic header information, such as the names of the sender and recipients, the subject, a timestamp, and a reference to the body and attachments of the message. The sender then requests the POST service to deliver this notification to each of the recipients.

It is noteworthy that ePOST extends recipient control beyond current systems by allowing the recipient to append the message to his mailbox or to simply ignore the notification, perhaps based on a spam filter. Since messages are stored in the sender organization's ring, one of the major goals of anti-spam researchers, to push most of the costs of spam back onto the spammers, can be achieved in a straightforward manner.

4.4 Email folders

Each mail folder is represented by a POST log. Each log entry represents a change to the state of the associated folder, such as the addition or deletion of a message. Furthermore, since the log can only be written by its owner and its contents are encrypted, ePOST preserves the expected privacy and integrity semantics of current email systems with storage on trusted servers.

An email insertion record contains the content of the message's MIME header, the message's handle and its decryption key, and a signature of all this information, taken from the sender's original notification message. This data is then encrypted under the user's public key.

4.5 Discussion

By default, ePOST provides strong confidentiality, authentication and message integrity. The system is able to tolerate up to $k-1$ faulty or unreachable nodes in any random set of k POST nodes without loss of data or service, where k is the degree of message replication. It relies on Pastry's secure routing facilities [2], data replication, and cryptographic techniques to achieve robustness under a wide range of attacks, including denial-of-service and participants that suffer byzantine faults.

More analysis and experimentation will be necessary to determine appropriate assumptions about the fraction of faulty nodes in various environments, and appropriate levels of replication. Results of a prior study on p2p filesystems in corporate environments indicate that modest levels of replication can yield high availability [1].

Since ePOST inserts all incoming messages into the local overlay, only the node failure probability and failure independence within a user's local overlay determine the durability of the messages that the user references. Therefore, a user's organization can take appropriate steps to ensure failure independence and determine an appropriate degree of replication.

Mailing lists can be easily supported by maintaining the list as an additional log and storing the log head reference at the list maintainer's user identity block. When delivering a message, the sender notices the list and expands the recipient list appropriately.

5 Incremental deployment

To allow an organization to adopt ePOST as its email infrastructure, ePOST must be able to interoperate with the existing email infrastructure. We sketch here how ePOST could be deployed in a single organization and interoperate with email services in the general Internet.

For inbound email, the organization's DNS server provides MX records referring to one of a set of POST nodes within the local organization. These nodes act as incoming SMTP mail gateways, accepting messages, inserting them into POST, and notifying the recipient's nodes. Suitable headers are generated such that the receiver is aware the message may have been transmitted on the Internet unencrypted. If no identity block can be found for the recipient in the local overlay, then the email "bounces" as in server-based systems.

The incoming mail gateway nodes need to be trusted to the extent that they receive plaintext email messages for local users. Typically, the desktop workstations of an organization's system administrators can be used for this purpose. These administrators own root passwords that allow them to access incoming email in conventional, server-based systems. Thus, ePOST provides the same privacy for incoming email from non-ePOST senders as existing systems.

Sending email to the outside world first requires determining that the desired email address is not already available inside the ePOST world. At that point, there

may be a gateway service that can provide the appropriate certificate material to generate a standard cryptographic email in S/MIME or PGP format. This encryption is performed in the sending user's local node, before the data goes onto the network. If the recipient does not support secure email, then the email must ultimately be transmitted in the clear, so the ePOST proxy server can speak regular SMTP to the recipient's mail server.

6 Related work

Lotus Notes and Microsoft Exchange provide a general, secure messaging infrastructure based on the client-server model, providing the ability to transfer email, personal contacts, calendars, and tasks. POST aims to provide similar functionality based on a serverless, decentralized and cooperative p2p architecture.

Current email protocols, including SMTP, POP3, and IMAP, are tailored towards an infrastructure based on dedicated servers. Minimal security is provided in these protocols, as they do not provide confidentiality, verifiability, or data integrity. Extensions like PGP provide secure email, but are not widely used.

The use of a single-writer, self-authenticating log in POST was inspired by the use of similar logs in the Ivy filesystem [11]. The loghead is the root of a Merkle hash tree [10], which allows the log to be stored on untrusted nodes, while ensuring that the authenticity of each log entry can be verified locally. This allows POST to avoid more complex byzantine state machine protocols [4].

A serverless email system proposed in [8] shares many of the goals of ePOST. Unlike POST, it focuses on email service only, and unlike ePOST, it is not compatible with the existing email infrastructure. Providing email services on top of a p2p storage system has also been explored in the OceanStore project [5]. The use of single-writer logs allows POST to achieve similar functionality with significantly less complexity, while providing general support for collaborative applications.

7 Status and conclusions

POST is a decentralized, collaborative messaging system that leverages the resources of participating desktop computers. POST provides highly resilient and scalable messaging services, while ensuring confidentiality, data integrity, and authentication. The basic services provided by POST can be used to support a variety of collaborative applications. In this paper, we have sketched how POST can be used to construct ePOST, a cooperative, secure email system.

Prototype implementations of POST and ePOST exist and are currently under experimental evaluation. Implementations of calendaring and instant messaging applications are underway. We plan to start using ePOST shortly, initially within our research groups, and hope to expand the user base within Rice and LIP6 and beyond, as we gain experience and confidence in the system. Given users' dependence on email services, we view this as a proof of concept for mission-critical p2p systems, and as a vehicle to gain practical experience and workload trace data from such a system.

References

[1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS'2000*, Santa Clara, CA, 2000.

[2] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proc. of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[3] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), October 2002.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[5] S. Czerwinski, A. Joseph, and J. Kubiatowicz. Designing a global email repository using OceanStore, June 2002. UC Berkeley summer retreat, `http://roc.cs.berkeley.edu/retreats/summer_02/slides/czerwin.pdf`.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, October 2001.

[7] J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS 2002)*, Vienna, Austria, July 2002.

[8] J. Kangasharju, K. Ross, D. Turner, J. Syrjala, and D.S. Digeon. Peer-to-peer e-mail, November 2002. Submitted for publication.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.

[10] R. Merkle. A digital signature based on a conventional encryption function. In *In Advances in Cryptology—CRYPTO'87 (LNCS, vol. 293)*, 1987.

[11] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.

[13] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, October 2001.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, August 2001.

[15] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. ITPTS'02*, Cambridge, MA, March 2002.

[16] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Crash-Only Software

George Candea and Armando Fox
Stanford University
{candea,fox}@cs.stanford.edu

Abstract

Crash-only programs crash safely and recover quickly. There is only one way to stop such software—by crashing it—and only one way to bring it up—by initiating recovery. Crash-only systems are built from crash-only components, and the use of transparent component-level retries hides intra-system component crashes from end users. In this paper we advocate a crash-only design for Internet systems, showing that it can lead to more reliable, predictable code and faster; more effective recovery. We present ideas on how to build such crash-only Internet services, taking successful techniques to their logical extreme.

1. Occam's Razor and the Restart Potpourri

There are many reasons to restart software, and many ways to do it. Studies have shown that a main source of downtime in large scale software systems is caused by intermittent or transient bugs [12, 20, 19, 1]. Most non-embedded systems have a variety of ways to stop; for example, an operating system can shut down cleanly, panic, hang, crash, lose power, etc.

When shutting down programs cleanly, unavailability consists of the time to shut down and the time to come back up; when crash-rebooting, unavailability consists only of the time to recover. Ironically, shutting down and reinitializing can sometimes take longer than recovering from a crash. Table 1 illustrates a casual comparison of reboot times; no important data was lost in either of the experiments.

System	Clean reboot	Crash reboot
RedHat 8 (with ext3fs)	104 sec	75 sec
JBoss 3.0 application server	47 sec	39 sec
Windows XP	61 sec	48 sec

Table 1. Duration of clean vs. crash reboots.

It is impractical to build a system that is guaranteed to never crash, even in the case of carrier class phone switches or high end mainframe systems. Since crashes are unavoidable, software must be at least as well prepared for a crash as it is for a clean shutdown. But then—in the spirit of Occam's Razor—if software is crash-safe, why support additional, non-crash mechanisms for shutting down? A frequent reason is the desire for higher performance.

For example, to avoid slow synchronous disk writes, many UNIX file systems cache metadata updates in memory. As a result, when a UNIX workstation crashes, the file system reaches an inconsistent state that takes a lengthy `fsck` to repair, an inconvenience that could have been avoided by shutting down cleanly. This captures the design tradeoff that improves steady state performance at the expense of shutdown and recovery performance. In the face of inevitable crashes, such a file system turns out to be brittle: a crash can lose data and, in some cases, the post-crash inconsistency cannot even be repaired. Not only do such performance tradeoffs impact robustness, but they also lead to complexity by introducing multiple ways to manipulate state, more code, and more APIs. The code becomes harder to maintain and offers the potential for more bugs—a fine tradeoff, if the goal is to build fast systems, but a bad idea if the goal is to build highly available systems. If the cost of such performance enhancements is dependability, perhaps it's time to reevaluate our design strategy.

In earlier work, we used recursive micro-reboots to improve the availability of a soft-state system that was trivially crash-safe [3]. In this paper we advocate a *crash-only design* (i.e., crash safety + fast recovery) for Internet systems, a class distinguished by the following properties: large scale, stringent high availability requirements, built from many heterogenous components, accessed over standard request-reply protocols such as HTTP, serving workloads that consist of large numbers of relatively short tasks that frame state updates, and subjected to rapid and perpetual evolution. We restrict our attention to single installations that reside inside one data center and do not span administrative domains.

In high level terms, a crash-only system is defined by the equations *stop=crash* and *start=recover*. In the rest of the paper, we describe the benefits of the crash-only design approach by analogy to physics, describe the internal properties of components in a crash-only system, the architectural properties governing the interaction of components, and a restart/retry architecture that exploits crash-only design, including our work to date on a prototype using J2EE.

2. Why Crash-Only Design ?

Mature engineering disciplines rely on macroscopic *descriptive* physical laws to build and understand the behavior of physical systems. These sets of laws, such as Newtonian mechanics, capture in simple form an observed physical invariant. Software, however, is an abstraction with no

physical embodiment, so it obeys no physical laws. Computer scientists have tried to use *prescriptive* rules, such as formal models and invariant proofs, to reason about software. These rules, however, are often formulated relative to an abstract model of the software that does not completely describe the behavior of the running system (which includes hardware, an operating system, runtime libraries, etc.). As a result, the prescriptive models do not provide a complete description of how the implementation behaves in practice, because many physically possible states of the complete system do not correspond to any state in the abstract model.

With the crash-only property, we are trying to impose, from outside the software system, macroscopic behavior that coerces the system into a simpler, more predictable universe with fewer states and simpler invariants. Each crash-only component has a single idempotent "power-off switch" and a single idempotent "power-on switch"; the switches for larger systems are built by wiring together their subsystems' switches in ways described by section 3. A component's power-off switch implementation is entirely external to the component, thus not invoking any of the component's code and not relying on correct internal behavior of the component. Examples of such switches include `kill -9` sent to a UNIX process, or turning off the physical, or virtual, machine that is running some software inside it.

Keeping the power-off switch mechanism external to components makes it a high confidence "component crasher." Consequently, every component in the system must be prepared to suddenly be deactivated. Power-off and power-on switches provide a very small repertoire of high-confidence, simple behaviors, leading to a small state space. Of course, the "virtual shutdown" of a virtual machine, even if invoked with `kill -9`, has a much larger state space than the physical power switch on the workstation, but it is still vastly simpler than the state space of a typical program hosted in the VM, and it does not vary for different hosted programs. Indeed, the fact that virtual machines are relatively small and simple compared to the programs they host has been successfully invoked as an argument for using VMs for inter-application isolation [26].

Recovery code deals with exceptional situations, and must run flawlessly. Unfortunately, exceptional situations are difficult to handle, occur seldom, and are not trivial to simulate during development; this often leads to unreliable recovery code. In crash-only systems, however, recovery code is exercised every time the system starts up, which should ultimately improve its reliability. This is particularly relevant given that the rate at which we reduce the number of bugs per Klines of code lags behind the rate at which the number of Klines per system increases, with the net result being that the number of bugs in an evolving system increases with time [7]. More bugs mean more failures, and systems that fail more often will need to recover more often.

Many of the benefits resulting from a crash-only design have been previously obtained in the data storage/retrieval world with the introduction of transactions. Our approach aims for a similar effect on the failure properties of Internet systems—crash-only design is in many ways a generalization of the transaction model. It is important to note that Internet applications do not have to use transactions in order to be crash-only; in fact, ACID semantics can sometimes lead to overkill. For example, session data accumulates information at the server over a series of user service requests, for use in subsequent operations. It is mostly single-reader/single-writer, thus not requiring ordering and concurrency control. The richness of a query language like SQL is unnecessary, and session state usually does not persist beyond a few minutes. These observations are leveraged by SSM [18], a crash-only hashtable-like session state store.

A crash-only system makes it affordable to transform every detected failure into component-level crashes; this leads to a simple fault model, and components only need to know how to recover from one type of failure. For example, [21] forced all unknown faults into node crashes, allowing the authors to improve the availability of a clusterized web server. Existing literature often assumes unrealistic fault models (e.g., that failures occur according to well-behaved tractable distributions); a crash-only design enables aggressive enforcement of such desirable fault models, thus increasing the impact of prior work. If we state invariants about the system's failure behavior and make such behavior predictable, we are effectively coercing reality into a small universe governed by well-understood laws.

Moreover, a system in which crash-recovery is cheap allows us to micro-reboot suspect components before they fail. By aggressively employing software rejuvenation [16]—rebooting in order to stave off failure induced by resource exhaustion—we can prevent failures altogether. The system can immediately trigger component-level rejuvenation whenever it notices fail-stutter behavior [2], a trough in offered workload, or based on predictive mathematical models of software aging [10].

Finally, if we admit that most failures can be recovered by micro-rebooting, crashing every suspicious component could shorten the fault detection and diagnosis time—a period that sometimes lasts longer than repair itself.

3. Properties of Crash-Only Software

In this section we describe a set of properties that we deem sufficient for a system to be crash-only. In some systems, some of these properties may not be necessary for crash-only behavior.

To make components crash-only, we require that all important non-volatile state be kept in dedicated state stores, that state stores provide applications with the right abstractions, and that state stores be crash-only. To make a system of interconnected components crash-only, it must be designed so that components can tolerate the crashes and temporary unavailability of their peers. This means we require strong modularity with relatively impermeable component

boundaries, timeout-based communication and lease-based resource allocation, and self-describing requests that carry a time-to-live and information on whether they are idempotent. Many Internet systems today have some subset of these properties, but we do not know of any that combines all properties into a true crash-only system.

In section 4 we will show how crash-only components can be glued together into a robust Internet system based on a restart/retry architecture; in the rest of this section we describe in more detail the properties of crash-only systems. Some relate to intra-component state management, while others relate to inter-component interactions. We recognize that some of these sacrifice performance, but we strongly believe the time has come for robustness to reclaim its status as a first-class citizen.

3.1. Intra-Component Properties

In today's Internet applications there are a small number of types of state: transactional persistent state, single-reader/single-writer persistent state (e.g., user profiles, that almost never see concurrent access), expendable persistent state (server-side information that could be sacrificed for the sake of correctness or performance, such as clickstream data and access logs), session state (e.g., the result set of a previous search, subject to refinement), soft state (state that can be reconstructed at any time based on other data sources), and volatile state. While differentiated mostly by guaranteed lifetime, the requirements for these categories of state lead to qualitatively different implementations.

All important non-volatile state is managed by dedicated state stores, leaving applications with just program logic. Specialized state stores (e.g., relational and object-oriented databases, file system appliances, distributed data structures [14], non-transactional hashtables [15], session state stores [18], etc.) are much better suited to manage state than code written by developers with minimal training in systems programming. Applications become stateless clients of the state stores, which allows them to have simpler and faster recovery routines. A popular example of such separation can be found in three-tier Internet architectures, where the middle tier is largely stateless and relies on backend databases to store data.

These state stores must also be crash-only, otherwise the problem has just moved down one level. Many commercial off-the-shelf state stores available today are crash-safe (i.e., they can be crashed without loss of data), such as databases and the various network-attached storage devices, but most are not crash-only, because they recover slowly. Many products, however, offer tuning knobs that permit the administrator to trade performance for improved recovery time, such as making checkpoints more often in the Oracle DBMS [17]. An example of a pure crash-only state store is the Postgres database system [25], which uses non-overwriting storage and maintains all data in a single append-only log. Although it trades away some performance, Postgres achieves practically instantaneous recovery, because it only needs to mark the uncommitted transactions as aborted.

The abstractions and guarantees provided by state stores must be congruent with application requirements. This means that the state abstraction exported by the state store is not too powerful (e.g., offering a SQL interface with ACID semantics for storing and retrieving simple key-value tuples) and not too weak. A state abstraction that is too weak will require client components to do some amount of state management, such as implementing a customer record abstraction over an offered file system interface. Good state abstractions allow applications to operate at their "natural" semantic level. Offering the weakest state guarantees that satisfy the application allows us to exploit application semantics and build simpler, faster, more reliable state stores.

For example, Berkeley DB [22] is a storage system supporting B+tree, hash, and record abstractions. It can be accessed through four different interfaces, ranging from no concurrency control/no transactions/no disaster recovery to a multi-user, transactional API with logging, fine-grained locking, and support for data replication. Applications can use the abstraction that is right for their purposes and the underlying state store optimizes its operation to fit those requirements. Workload characteristics can also be leveraged by state stores; e.g., expecting a read-mostly workload allows a state store to utilize write-through caching, which can significantly improve recovery time and performance.

We do not advocate that every application have its own set of state stores. Instead, we believe Internet systems will standardize on a small number of state store types: ACID stores (e.g., databases for customer and transaction data), non-transactional persistent stores (e.g., DeStor [15], a crash-only system specialized in handling non-transactional persistent data, like user profiles), session state stores (e.g., SSM [18] for shopping carts), simple read-only stores (e.g., file system appliances for static HTML and images), and soft state stores (e.g., web caches). If we think carefully about the state abstractions required by each application component and use suitable state stores, we can make these components crash-only.

3.2. Inter-Component Properties

Subsystems that crash on their own, or that are explicitly crash-rebooted for recovery, will temporarily become unavailable to serve requests. For a crash-only system to gracefully tolerate such behavior, we need to decouple components from each other, from the resources they use, and from the requests they process.

Components have externally enforced boundaries that provide strong fault containment. The desired isolation can be achieved with virtual machines, isolation kernels [26], task-based intra-JVM isolation [24, 8], OS processes, etc. Indeed, the Denali isolation kernel is designed for such en-

capsulation. Web hosting service providers often use multiple virtual machines on one physical machine to offer their clients individual web servers they can administer at will, without affecting other customers. The boundaries between components delineate distinct, individually recoverable stages in the processing of requests.

All interactions between components have a timeout. This includes explicit communication as well as RPC: if no response is received to a call within the allotted timeframe, the caller assumes the callee has failed and reports it to a recovery agent [5], which can crash-restart the callee if appropriate. Crash-restarting helps ensure the called component is in a known state; this is safe because the component is crash-safe and crash-restart is idempotent. Timeouts provide an orthogonal mechanism for turning all non-Byzantine failures, both at the component level and at the network level, into fail-stop events (i.e., the failed entity either provides results or is stopped), even though the components are not necessarily fail-stop. Such behavior is easier to accomodate, and containment of faults is improved.

Crash-only components recover quickly, thus recovery is very cheap. Under such circumstances, it becomes acceptable for the recovery manager to crash-recover suspect components even when it lacks the certainty that those components have indeed failed; the downtime risk of letting the components run may be higher than crash-rebooting healthy components.

All resources are leased, rather than permanently allocated, to ensure that resources are not coupled to the components using them. Resources include many types of persistent state, such as account profiles for a free e-mail provider: every time the user logs in, a 6-month lease is renewed; when the lease expires, all associated data can be purged from the system. It also includes CPU resources: if a computation is unable to renew its execution lease, it is terminated by a high confidence watchdog [9]. For example, in PHP, a server-side scripting language used for writing dynamic web pages, runaway scripts are killed and an error is returned to the web browser. Leases [11] give us the ability to reason about the conditions that hold true of the system's resources after a lease expires. Infinite timeouts or leases are not acceptable; the maximum-allowed timeout and lease are specified in an application-global policy. This way it is less likely that the system will hang or become blocked.

Requests are entirely self-describing, by making the state and context needed for their processing explicit. This allows a fresh instance of a rebooted component to pick up a request and continue from where the previous instance left off. Requests also carry information on whether they are idempotent, along with a time-to-live; both idempotency and TTL information can initially be set at the system boundary, such as in the web tier. For example, the TTL may be determined by load or service level agreements, and idempotency flags can be based on application-specific information (which can be derived, for instance, from URL substrings that determine the type of request). Many interesting operations in an Internet service are idempotent, or can easily be made idempotent by keeping track of sequence numbers or by wrapping requests in transactions; some large Internet services have already found it practical to do so [23]. Over the course of its lifetime, a request will split into multiple sub-operations, which may rejoin, in much the same way nested transactions do. Recovering from a failed idempotent sub-operation entails simply reissuing it; for non-idempotent operations, the system can either roll them back, apply compensating operations, or tolerate the inconsistency resulting from a retry. Such transparent recovery of the request stream can hide intra-system component failures from the end user.

4. A Restart/Retry Architecture

A component infers failure of a peer component either based on a raised exception or a timeout. When a component is reported failed, a recovery agent may crash-reboot it; the idempotency of crash-shutdown makes this an inexpensive way to ensure the component is indeed turned off before attempting recovery. Components waiting for an answer from the restarted component receive a *RetryAfter(n)* exception, indicating that the in-flight requests can be resubmitted after n msec (the estimated time to recover); this exception is purely an optimization because, in its absence, components would have timeouts as a fallback mechanism. If the request is idempotent and its time-to-live allows it to be resubmitted, then the requesting component does so. Otherwise, a failure exception is propagated up the request chain until either a previous component decides to resubmit, or the client needs to be notified of the failure. The web front-end issues an HTTP/1.1 `Retry-After` directive to the client with an estimate of the time to recover, and retry-capable clients can resubmit the original HTTP request.

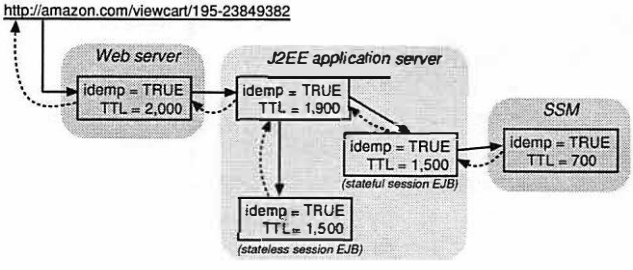


Figure 1. A simple restart/retry architecture.

In Figure 1 we show a simple restart/retry example, in which a request to view a shopping cart splits inside the application server into one subrequest to a stateful session EJB (Enterprise JavaBean) that communicates with a session state store and another subrequest to a stateless session EJB. Should the state store become unavailable, the application server either receives a *RetryAfter* exception or times

out, at which time it can decide whether to resubmit the request to SSM or not. Within each of the subsystems shown in Figure 1, we can imagine each subrequest further splitting into finer grain subrequests submitted to the respective subsystems' components. We have implemented crash-restarting of EJBs in a J2EE application server; an EJB-level micro-reboot takes less than a second [5].

Timeout-based failure detection is supplemented with traditional heartbeats and progress counters. The counters—compact representations of a component's processing progress—are usually placed at state stores and in messaging facilities, where they can map state access and messaging activity into per-component progress. Many existing performance monitors can be transformed into progress monitors by augmenting them with request origin information. Components themselves can also implement progress counters that more accurately reflect application semantics, but they are less trustworthy, because they are inside the components.

The dynamics of loosely coupled systems can sometimes be surprising. For example, resubmitting requests to a component that is recovering can overload it and make it fail again; for this reason, the *RetryAfter* exceptions provide an estimated time-to-recover. This estimated value can be used to spread out request resubmissions, by varying the reported time-to-recover estimate across different requestors. A maximum limit on the number of retries is specified in the application-global policy, along with the lease durations and communication timeouts. These numbers can be dynamically estimated based on historical information collected by a recovery manager [5], or simply captured in a static description of each component, similar to deployment descriptors for EJBs. In the absence of such hints, a simple load balancing algorithm or exponential backoff can be used.

To prevent reboot cycles and other unstable conditions during recovery, it is possible to quiesce the system when a set of components is being crash-rebooted. This can be done at the communication/RPC layer, or for the system as a whole. In our prototype, we use a stall proxy [5] in front of the web tier to keep new requests from entering the system during the recovery process. Since Internet workloads are typically made of short running requests, the stall proxy transforms brief system unavailability into temporarily higher latency for clients. We are exploring modifications to the Java RMI layer that would allow finer grain request stalling.

5. Discussion

Building crash-only systems is not easy; the key to widespread adoption of our approach will require employing the right architectural models and having the right tools. With the recent success of component-based architectures (e.g., J2EE and .Net), and the emergence of the application server as an operating system for Internet applications, it is possible to provide many of the crash-only properties in the platform itself. This would allow all applications running on that platform to take advantage of the effort and become crash-only.

We are applying the principles described here to an open-source Java 2 Enterprise Edition (J2EE) application server. We are separating the individual J2EE services (naming, directory lookup, messaging, etc.) into well-isolated components, implementing requests as self-describing continuations, modifying the RMI layer to allow for timeout-based operation, modifying the EJB containers to implement lease-based resource allocation, and integrating non-transactional state stores like DeStor and SSM. A first step in this direction is described in [5].

We are focusing initially on applications whose workloads can be characterized as relatively short-running tasks that frame state updates. Substantially all Internet services fit this description, in part because the nature of HTTP has forced designers into this mold. As enterprise services and applications (e.g., workflow, customer management) become web-enabled, they adopt similar architectures. We expect there are many applications outside this domain that could not easily be cast this way, and for which deriving a crash-only design would be impractical or infeasible.

In order for the restart/retry architecture to be highly available and correct, most requests it serves must be idempotent. This requirement might be inappropriate for some applications. Our proposal does not handle Byzantine failures or data errors, but such behavior can be turned into fail-stop behavior using well-known orthogonal mechanisms, such as triple modular redundancy [13] or clever state replication [6].

In today's Internet systems, fast recovery is obtained by overprovisioning and counting on rapid failure detection to trigger failover. Such failover can sometimes successfully mask hours-long recovery times, but often detecting failures end-to-end takes longer than expected. Crash-only software is complementary to this approach and can help alleviate some of the complex and expensive management requirements for highly redundant hardware, because faster recovering software means less redundancy is required. In addition, a crash-only system can reintegrate recovered components faster, as well as better accommodate removed, added, or upgraded components.

We expect throughput to suffer in crash-only systems, but this concern is secondary to the high availability and predictability we expect in exchange. The first program written in a high-level language was certainly slower than its hand-coded assembly counterpart, yet it set the stage for software of a scale, functionality and robustness that had previously been unthinkable. These benefits drove compiler writers to significantly optimize the performance of programs written in high-level languages, making it hard to imagine today how we could program otherwise. We expect the benefits of crash-only software to similarly drive efforts that will erase, over time, the potential performance loss of such designs.

6. Conclusion

By using a crash-only approach to building software, we expect to obtain better reliability and higher availability in Internet systems. Application fault models can be simplified through the application of externally-enforced "crash-only laws," thus encouraging simpler recovery routines which have higher chances of being correct. Writing crash-only components may be harder, but their simple failure behavior can make the assembly of such components into large systems easier.

The promise of a simple fault model makes stating invariants on failure behavior possible. A system whose component-level and system-level invariants can be enforced through crash-rebooting is more predictable, making recovery management more robust. It is our belief that applications and services with high availability requirements can significantly benefit from these properties.

Once we surround a crash-only system with a suitable recovery infrastructure, we obtain a recursively restartable system [4]. Transparent recovery based on component-level micro-reboots enables restart/retry architectures to hide intra-system failure from the end users, thus improving the perceived reliability of the service. We find it encouraging that our initial prototype [5] was able to complete 78% more client requests under faultload than a non-crash-only version of the system that did not employ micro-reboots for recovery.

References

[1] T. Adams, R. Igou, R. Silliman, A. M. Neela, and E. Rocco. Sustainable infrastructures: How IT services can address the realities of unplanned downtime. Research Brief 97843a, Gartner Research, May 2001. Strategy, Trends & Tactics Series.

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001.

[3] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive micro-reboots: A soft-state system case study. *Performance Evaluation Journal*, Summer 2003. To appear.

[4] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001.

[5] G. Candea, P. Keyani, E. Kiciman, S. Zhang, and A. Fox. JAGR: An autonomous self-recovering application server. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999.

[7] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Lake Louise, Canada, 2001.

[8] G. Czajkowski and L. Daynés. Multitasking without compromise: A virtual machine evolution. In *Proc. Conference on Object Oriented Programming Systems Languages and Applications*, Tampa Bay, FL, 2001.

[9] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb. 2003.

[10] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi. A methodology for detection and estimation of software aging. In *Proc. 9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[11] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, 1989.

[12] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[13] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[14] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[15] A. C. Huang and A. Fox. Decoupled storage: State with stateless-like properties. Submitted to the 22nd Symposium on Reliable Distributed Systems, 2003.

[16] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995.

[17] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: Quick fault recovery in Oracle. In *Proc. ACM International Conference on Management of Data*, Santa Barbara, CA, 2001.

[18] B. Ling and A. Fox. A self-tuning, self-protecting, self-healing session state management layer. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, 2003.

[19] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proc. 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, 1999. Tutorial.

[20] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11:341–353, 1995.

[21] K. Nagaraja, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault model enforcement to improve availability. In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[22] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer USENIX Technical Conference*, Monterey, CA, June 1999.

[23] A. Pal. Personal communication. Yahoo!, Inc., 2002.

[24] P. Soper, P. Donald, D. Lea, and M. Sabin. Application isolation API specification. Java Specification Request No. 121, `http://jcp.org/en/jsr/detail?id=121`, 2002.

[25] M. Stonebraker. The design of the Postgres storage system. In *Proc. 13th Conference on Very Large Databases*, Brighton, England, 1987.

[26] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

The *Phoenix* Recovery System: Rebuilding from the ashes of an Internet catastrophe

Flavio Junqueira Ranjita Bhagwan Keith Marzullo Stefan Savage Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego

1 Introduction

The Internet today is highly vulnerable to *Internet catastrophes*: events in which an exceptionally successful Internet pathogen, like a worm or email virus, causes data loss on a significant percentage of the computers connected to the Internet. Incidents of successful wide-scale pathogens are becoming increasingly common on the Internet today, as exemplified by the Code Red and related worms [6], and LoveBug and other recent email viruses [11]. Given the ease with which someone can augment such Internet pathogens to erase data on the hosts that they infect, it is only a matter of time before Internet catastrophes occur that result in large-scale data loss.

In this paper, we explore the feasibility of using data redundancy, a model of dependent host vulnerabilities, and distributed storage to tolerate such events. In particular, we motivate the design of a cooperative, distributed remote backup system called the *Phoenix* recovery system. The usage model of *Phoenix* is straightforward: a user specify an amount F of bytes from its disk space the system can use, and the goal of the system is to protect a proportional amount F/k of its data using storage provided by other hosts.

In general, to recover the lost data of a host that was a victim in an Internet catastrophe, there must be copies of that data stored on a host or set of hosts that survived the catastrophe. A typical replication approach [10] creates t additional replicas if up to t copies of the data can be lost in a failure. In our case, t would need to be as large as the largest Internet catastrophe. As an example, the Code Red worm infected over 359,000 computers, and so t would need to be larger than 359,000 for hosts to survive a similar kind of event. Using such a large degree of replication would make cooperative remote backup useless for at least two reasons. First, the amount of data each user can protect is inversely proportional to the degree of replication, and with such a vast degree of replication the system could only protect a minuscule amount of data per user. Second, ensuring that such a large number of replicas are written would take an impractical amount of time.

Our key observation that makes *Phoenix* both feasible and practical is that an Internet catastrophe, like any large-scale Internet attack, exploits shared vulnerabilities. Hence, users should replicate their data on hosts that do not have the same vulnerabilities. That is, the replication mechanism should take the dependencies of host failures—in this case, host diversity—into account [5]. Hence, we formally represent host attributes, such as its operating system, web browser, mail client, web server, etc. The system can then use the attributes of all hosts in the system to determine how many replicas are needed to ensure recoverability, and on which hosts those replicas should be placed, to survive an Internet catastrophe that exploits one of its attributes. For example, for hosts that run a Microsoft web server, the system will avoid placing replicas on other hosts that run similar servers so that the replicas will survive Internet worms that exploit bugs in the server. Such a system could naturally be extended to tolerate simultaneous catastrophes using multiple exploits, although at the cost of a reduced amount of recoverable data that can be stored. Using a simulation model we show that, by doing informed placement of replicas, a *Phoenix* recovery system can provide highly resilient and available cooperative backup with low overhead.

In the rest of this paper, we discuss various approaches for tolerating Internet catastrophes and motivate the use of a cooperative, distributed recovery system like *Phoenix* for surviving them. Section 3 then describes our model for dependent failures and how we apply it to tolerate catastrophes. In Section 4, we explore the design space of the amount of available storage in the system and the redundancy required to survive Internet catastrophes under various degrees of host diversity and shared vulnerabilities. We then discuss system design issues in Section 5. Finally, Section 6 concludes the paper.

2 Motivation

Backups are a common way to protect data from being lost as a result of a catastrophe. We know of three approaches to backup.

Local backup is the most common approach for recovering from data loss, and it has many advantages. Users and organizations have complete control over the amount and frequency with which data is backed up. Furthermore, tape and optical storage are inexpensive, high capacity devices. However, large organizations that have

large amounts of data have to employ personnel to provide the backup service. Individual home users often do not use it because of the time and hassle of doing so, causing home systems to be highly vulnerable to exploit and potential data loss.

Another approach is to use a *commercial remote backup* service, such as DataThought Consulting [4] or Protect-Data.com [9]. This approach is convenient, yet expensive. Currently, automatic backup via a modem or the Internet for 500MB of data costs around $30-$125 a month.

Cooperative remote backup services provide the convenience of a commercial backup service but at a more attractive price. Instead of paying money, users relinquish a fraction of their computing resources (disk storage, CPU cycles for handling requests, and network bandwidth for propagating data). pStore [1] is an example of such a service. However, its primary goal is to tolerate local failures such as disk crashes, power failures, etc. Pastiche [2] also provides similar services, while trying to minimize storage overhead by finding similarities in data being backed up. Its aim is also to guard against localized catastrophes, by storing one replica of all data in a geographically remote location.

We believe that a cooperative, distributed system is a compelling architecture for providing a convenient and effective approach for tolerating Internet catastrophes. It would be an attractive system for individual Internet users, like home broadband users, who do not wish to pay for commercial backup service or do not want the hassle of making their own local backups frequently. Users of *Phoenix* would not need to exert any significant effort to backup their data. Specifying what data to protect can be made as easy as specifying what data to share on a file sharing peer-to-peer system. Further, a cooperative architecture has little cost in terms of time and money; instead, users relinquish a small fraction of their computer resources to gain access to a highly resilient backup service. A user specifies an amount F of bytes from its disk space to be used by the system, and the system would protect a proportional amount F/k of its data. We observe that the value k depends on the host diversity, and can differ among the hosts. In addition, the system would limit the network bandwidth and CPU utilization to minimize the impact of the service on normal operation.

To our knowledge, *Phoenix* is the first effort to build a cooperative backup system resilient to wide-scale Internet catastrophes.

3 Taking Advantage of Diversity

Traditionally, reliable distributed systems are designed using the threshold model: out of n components, no more than $t < n$ are faulty at any time. Although this model can always be applied when the probability of having a total failure is negligible, it is only capable of expressing the worst-case failure scenario, and it is best suited when failures are independent and identically distributed. The worst-case, however, can be one in which the failures of components are highly correlated.

Failures of hosts in a distributed system can be correlated for several reasons. Hosts may run the same code or be located in the same room, for example. In the former case, if there is a vulnerability in the code, then it can be exploited in all the hosts executing the target software. In the latter case, a power outage can crash all machines plugged into the same electrical circuit.

As a first step towards the design of a cooperative backup system for tolerating catastrophes, we need a concise way of representing failure correlation. We use the *core* abstraction to represent correlation among host failures [5]. A core is a reliable minimal subset of components: the probability of having all hosts in a core failing is negligible, for some definition of negligible. In a backup system, a core corresponds to the minimal replica set required for resilience.

Determining the cores of a system depends on the failure model used and the desired degree of resilience for the system. The failure model prescribes the possible types of failures for components. These types of failures determine how host failures can be correlated. In our case, hosts are the components of interest and software vulnerabilities are the causes of failures. Consequently, hosts executing the same piece of software present high failure correlation. This information on failure correlation is not sufficient, however, to determine the cores of a system. It also depends on the desired degree of resilience. As one increases the degree of resilience, more components are perhaps necessary to fulfill the core property stated above.

To reason about the correlation of host failures, we associate attributes to hosts. The attributes represent characteristics of the host that can make it prone to failures. For example, the operating system a host runs is a point of attack: an attack that targets Linux is less likely to be effective against hosts running Solaris, and is even less effective against hosts running Windows XP. We could represent this point of attack by having an n-ary attribute that indicates the operating system, where the value of the attribute is 0 for Linux, 1 for Windows XP, 2 for Solaris, and so on. Throughout this paper, we use A as the set of attributes that characterize a host.

To illustrate the concepts introduced in this section, consider the system described in Example 3.1. In this system, hosts are characterized by three attributes and each attribute has two possible values. We assume that hosts fail due to crashes caused by software vulnerabilities, and at most one vulnerability can be exploited at a time. Note

that the cores shown in the example have maximum resilience according to the given set of attributes.

Example3.1 :

Attributes: $Operating\,System = \{Unix, Windows\}$;
$Web\ Server = \{Apache, IIS\}$;
$Web\ Browser = \{IE, Netscape\}$.

Hosts: $H_1 = \{Unix, Apache, Netscape\}$;
$H_2 = \{Windows, IIS, IE\}$;
$H_3 = \{Windows, IIS, Netscape\}$;
$H_4 = \{Windows, Apache, IE\}$.

Cores $= \{H_1H_2, H_1H_3H_4\}$.

There are a few interesting facts to be observed about Example 3.1. First, H_1 and H_2 form what we call an *orthogonal core*, which is a core composed of hosts that have different values for every attribute. Note that in this case the size of the orthogonal core is two because of our assumption that at most one vulnerability can be exploited at a time. This implies that it is necessary and sufficient to have two hosts with different values for every attribute. Even though it is not orthogonal, $H_1H_2H_3$ is also a core since it covers all attributes. Second, when choosing a core for host H_1 to store replicas of its data, there are two possibilities: H_1H_2 and $H_1H_3H_4$. The second option for a core is larger than the first. Thus, choosing the second leads to unnecessary replication. The optimal choice in terms of storage overhead is therefore H_1H_2.

Choosing a smallest core may seem a good choice at first because it requires fewer replicas. We observe, however, that such a choice can adversely impact the system. In environments with highly skewed diversity, the total capacity of the system may be impacted by always choosing the smallest core[1]. Back in Example 3.1, H_1 is the only host which has some flavor of Unix as the operating system. Consequently, a core for every other host has to contain H_1. For a small system as the one in the example this should not be a problem, but it is a potential problem for large-scale deployments. This raises the question of how host diversity impacts on storage overhead, storage load, and resilience. We address this question in the next section.

4 Host Diversity

We now develop a metric for specifying attribute diversity among a set of hosts, and a system model for representing sets of hosts with various degrees of host diversity. We then use this model to quantify the core sizes, and hence the amount of replication, required to achieve high degrees of resilience to Internet catastrophes under a wide range of diversities of host vulnerabilities.

[1] By skewed diversity, we mean a distribution of attribute configurations that is not uniform.

4.1 Diversity and Core Sizes

If one knew the probability of attack for each vulnerability, then given a target system resilience one could enumerate cores with that target resilience. In our case, it is not clear how one would determine such probabilities. Instead, we define a core c for a host p to be a minimal set of hosts with the following additional properties: 1) $p \in c$; 2) for every attribute $a \in A$, either there is a host in c that differs from p in the value of a or there is no host in the system that differs from p in the value of a. Such a subset of hosts is a core for a host p if we assume that, in any Internet catastrophe, an attack targets a single attribute value. Although it is not hard to generalize this definition to allow for attacks targeted against multiple attribute values, in the rest of this paper we focus on attacks against a single attribute value.

Smaller cores means less replication, which is desirable for reducing storage overhead. A core will contain between 2 and $|A| + 1$ hosts. If the hosts' attributes are well distributed, then the cores will be small on average: for any host p, it is likely that there is a host q that has different values of each of the attributes, and so p and q constitute an orthogonal core. That is, a fair number of orthogonal cores are likely to exist. If there is less diversity, though, then the smallest cores may not be orthogonal for many hosts, thus increasing storage overhead.

A lack of diversity, especially when trying to keep core sizes small, can lead to a more severe problem. Suppose there are k hosts $\{p_1, p_2, \ldots p_k\}$ and an attribute a such that all have the same value for a. Moreover, there is only one host p that differs in the value of a. A core for each host p_i hence contains p, meaning that p will maintain copies for all of the p_i. Since the amount of disk space p donates for storing backup data is fixed, each p_i can only use $1/k$ of this space. In other words, if p donates F bytes for common storage to the system, then each p_i can back up only F/k bytes. Note that k can be as large as the number of hosts, and so F/k can be minuscule. In Example 3.1, host H_1 is the only one to have a different value for attribute "Operating System", and hence has to store copies for all the other hosts.

Characterizing the diversity of a set of hosts is a challenging task. In particular, considering all possible distributions for attribute configurations is not feasible. Instead, we define a measure f that condenses the diversity of a system into a single number. According to our definition, a system with diversity f is one in which a share f of the servers is characterized by a share $(1 - f)$ of the combinations of attributes. Although this metric is coarse and does not capture all possible scenarios, it is expressive enough to enable one to observe how the behavior of a backup system is affected by skewed diversity. Note that f is in the interval $[0.5, 1)$. The value $f = 0.5$ corre-

sponds to a uniform distribution, and a value of f close to 1 indicates a highly skewed diversity.

We use this metric to study how storage overhead, storage load, and resilience vary with skew in diversity. We define the storage overhead of a host p as the size of the core that p uses to backup its data. Host p maintains copies for k other hosts. We define the storage load of p to be such a value k. Note that storage load may vary among the hosts. Thus, we define the storage load of the system as the maximum value of k across all the hosts. In the remainder of this paper, we refer to the storage load of the system as just storage load. Finally, resilience depends on the number of attributes covered in a core, and it decreases as the number of non-covered attributes increases. We then define the resilience of the system for a host p as the percentage of attributes covered by the core c that p uses to backup its data.

The problem of finding a smallest core given a set of hosts and an attribute configuration for each host, however, is NP-hard (reduction from SET-COVER). For this reason, we used a randomized heuristic to find cores. This heuristic finds a core for a host p as follows:

1. It tries to find other hosts that have a fully disjoint set of attributes. If there is more than one host, then it picks one randomly;

2. If it finds no host in the previous step, then it randomly chooses hosts that have at least one different attribute until a core is constructed or there are no hosts left to choose.

This heuristic may not be the best; We have not yet done a thorough study of heuristics for finding cores. The results we present below, however, indicates that it is efficient in terms of storage overhead and resilience.

4.2 Modeling Diversity

To better understand the impact of diversity skew, we simulate a system of hosts with various attributes . On the Internet, most hosts run some version of Windows with Internet Explorer as the web browser [8], so we biased the attribute distribution towards having some fixed subset of attributes. The size of this subset depends on the value f chosen for the diversity of the system. To see this, consider a subset of size α. Assuming that each attribute has y possible values, for such a subset the total number of distinct configurations is $y^{|A|-\alpha}$. Thus, there is some integer α, $\alpha \leq |A|$, that satisfies the following equation:

$$\frac{\frac{y^{|A|-\alpha}}{y^{|A|}} \geq (1-f) > \frac{y^{|A|-(\alpha+1)}}{y^{|A|}}}{\rightarrow \quad \frac{1}{y^{\alpha}} \geq (1-f) > \frac{1}{y^{\alpha+1}}.} \qquad (1)$$

In our simulations, we compute the value of α using Equation 1, and then pick a subset $A' \subseteq A$ of attributes such that $|A'| = \alpha$. For every attribute a in A', we fix the value of a for a fraction f of the hosts. We then randomly choose values for the remaining attributes for this fraction of hosts. For the remaining hosts, we pick attribute configurations at random, but we make sure that each configuration is not a configuration of any host in the first fraction.

4.3 Simulation Results

Figures 1, 2, and 3 show the results of our simulations. We simulate a system of 1,000 hosts and present results for two scenarios: 8 attributes with 2 values each (8/2) and 8 attributes with 4 values each (8/4). The choice of 8 attributes is based upon an examination of the most targeted categories of software from public vulnerability databases, such as [7, 11]. From these databases, we observed 8 significant software categories and chose this value as a reasonable parameter for our simulations.

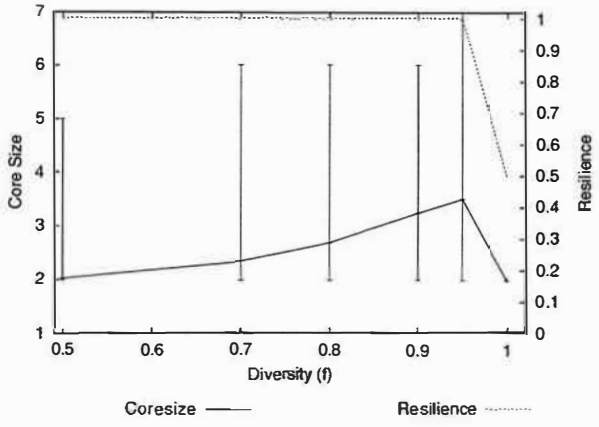


Figure 1: Core sizes as a function of diversity for 8 attributes, 2 values each.

We chose two different numbers of values per attribute to 1) explore a worst case and 2) show how core size and storage load benefit as a result of more fine-grained attribute configurations. The choice of 2 values per attribute corresponds to the coarsest division of hosts possible (e.g., Windows vs. Linux), and represents the worst case in terms of core size and storage load. We note that no vulnerabilities exploited by Internet pathogens have been so extreme, and that pathogens tend to exploit vulnerabilities at finer attribute granularities (e.g., Code Red and its variants exploited a vulnerability in Microsoft IIS running on Windows NT). The choice of 4 values per attribute represents a more fine-grained attribute configurations, and we use it to demonstrate how such configurations significantly improve core sizes and reduce storage load.

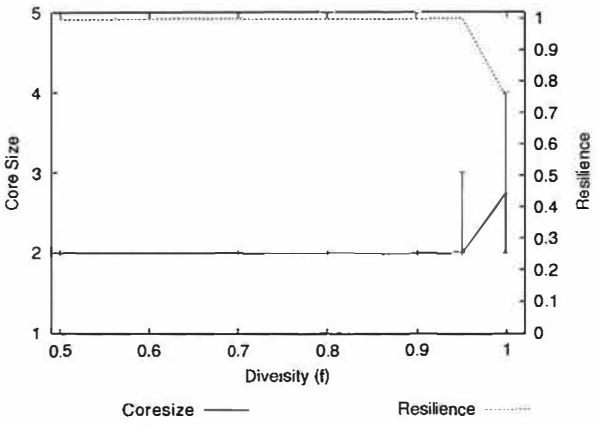


Figure 2: Core sizes as a function of diversity for 8 attributes, 4 values each.

We only show the results for one sample generated for each value of f, as we did not see significant variation across samples. Figures 1 and 2 show the core size averaged over cores for all of the hosts for different values of the diversity parameter f. We also include a measure of resilience that shows whether our algorithm was able to cover all attributes or not. A point in the resilience curve is hence the number of covered attributes, averaged over all hosts, divided by the total number of attributes. Note that resilience 1.0 means that all attributes are covered.

To show the variability in core size, we include error bars in the graphs showing the maximum and the minimum core sizes for values of f. The variability in core size is noticeably high in the 8/2 scenario, whereas it is lower in the 8/4 scenario. Because there are more configurations available in the 8/4 setting, it is likely that a host p finds a host q which has different values for every attribute even when the diversity is highly skewed.

Regarding the average core size, in the 8/2 scenario, it remains around 2 for values of f under 0.7, and goes up to average sizes around 3 for higher values of f. In either case, storage overhead is low, although it is overall higher than the average core size for the 8/4 scenario. The result of adding more attribute values to each attribute is therefore a reduction in storage overhead. In this scenario, the average core size remains around 2 for most of the values of f. It only increases for $f = 0.999$.

It is important to note that there is a drop in resilience for $f = 0.999$ in both scenarios. Observe that, for such a value of f, there are 999 hosts sharing some subset A' of attributes with a fixed value for each attribute and a single host p not sharing this subset. As a consequence, host p has to be in the core of a host q sharing A'. Host p, however, may not cover all the attributes of q. This being the case, there are possibly other hosts that cover the remaining attributes of q that p does not cover. If there are no such hosts, then there is no core for q which covers all attributes. The resilience for this host is therefore lower.

An important question that remains to be addressed is how much backup data a host will need to store. We address this question with the help of Figure 3. In this figure, the y-axis plots storage load. Thus, if $y = 10$, for example, there is a host p that must be in 10 cores given the core compositions that we computed, and every other host must be in 10 or fewer cores. As expected, storage load increases as f approaches 1, and reaches 1,000 for $f = 0.999$ in both scenarios. This is due to our previous observation that, for this value of f, there is a single host which has to be in the core of every other host. We conclude that the storage overhead for such a highly skewed diversity is small, but the total load incurred in a small percentage of the hosts can be very high.

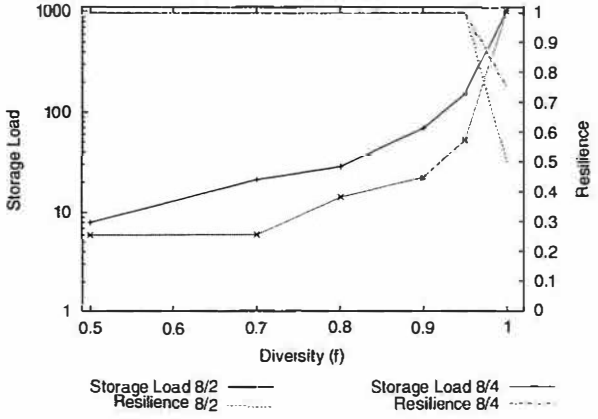


Figure 3: Storage load as a function of diversity for the 8/2 and the 8/4 scenarios.

Although we have presented results only for 1,000 hosts, we have also looked into other scenarios with a larger number of hosts. For 10,000 hosts and the same attribute scenarios, there is no reduction in resilience, and the average core size remains in the same order of magnitude. As we add more hosts to the system, we increase the probability of a host having some particular configuration, thus creating more possibilities for cores. The trend for storage load is the same as before: the more skewed the distribution of attribute configurations, the higher the storage load. For highly skewed distributions and large number of hosts, storage load can be extremely high. One important observation, however, is that as the population of hosts in the system increases, the number of different attribute configurations and the number of hosts with some particular configuration are likely to increase. Thus, for some scenario and fixed value of f, storage load does not increase linearly with the number of hosts. In our diversity model, it actually remains in the same order of magnitude.

Suppose now that we want to determine a bound on f

for a real system given our preliminary results. According to [8], over 93% of the hosts that access a popular web site run some version of Internet Explorer. This is the most skewed distribution of software they report (the second most skewed distribution is the percent of hosts running some version of Windows, which is 90%). There are vulnerabilities that attack all versions of Internet Explorer [11], and so f for such a collection of hosts can be no larger than 0.93. Note that as one adds attributes that are less skewed, they will contribute to the diversity of the system and reduce f.

In the lists provided by [8], there are 14 web browsers and 11 operating systems. For an idea of how a scenario like this would behave, consider a system of 1,000 hosts with 2 attributes and 14 values per attribute. For a value of $f = 0.93$ we have an average core size of 2, a maximum core size of 2, and storage load of 24. We did not see significant changes in these values when changing the number of values per attribute from 14 to 11.

A storage load of 24 means that there is some host that has to store backup data from 24 other hosts, or 4% of its storage to each host. We observe that this value is high because our heuristic optimizes for storage overhead. In an environment with such a skewed diversity, a good heuristic will have to take into account not only storage overhead, but the storage load of available hosts as well.

5 System Design Issues

The previous section gives us an idea of how much replication and how much storage is required in *Phoenix*. We end by briefly mentioning a number of design issues that an implementation of *Phoenix* needs to address as well.

The heuristics used for core identification need to use an index that maps hosts to the different attributes they possess. *Phoenix* therefore needs to maintain this index, which we intend to implement using a distributed hash table (DHT). Once *Phoenix* has identified a core, it stores copies of data on the hosts in the core. To ensure the integrity of the data, we plan on using some encryption mechanism. Thus, data is encrypted before releasing it to the hosts of a core. As observed in the previous section, it is also necessary to ensure fairness of storage allocation across users. For this, our heuristic to find cores will have to be modified to take storage load into account. Finally, we need to more carefully model the set of vulnerabilities and allow for dynamically adding and removing attributes/values.

In the wake of an Internet catastrophe, *Phoenix* itself has to continue functioning satisfactorily. Since we intend to use a DHT as a platform, it will need to survive a scenario where a large number of hosts suddenly leave the system [3]. Moreover, once there is a catastrophe, many users may try to recover files at the same time, potentially overloading the system; since recovery time is not critical, a distributed scheduler using randomized exponential wait times can ease recovery demand.

We are currently working on addressing these issues in a prototype design and implementation of *Phoenix*.

6 Conclusions

In this paper, we have explored the feasibility of using a cooperative remote backup system called *Phoenix* as an effective approach for surviving Internet catastrophes. *Phoenix* uses data redundancy, a model of dependent host failures, and distributed storage in a cooperative system. Using a simulation model we have shown that, by performing informed placement of replicas, *Phoenix* can provide highly reliable and available cooperative backup and recovery with low overhead.

References

[1] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Unpublished report, Dec. 2001.

[2] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[3] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, Oct. 2001.

[4] Datathought website, http://www.datathought.com.

[5] F. Junqueira and K. Marzullo. Synchronous Consensus for dependent process failures. In *Proceedings of the ICDCS 2003*, pages 274–283, May 2003.

[6] D. Moore, C. Shannon, and J. Brown. Code-Red: A case study on the spread and victims of an Internet worm. In *Proceedings of the 2002 ACM SICGOMM Internet Measurement Workshop*, pages 273–284, Marseille, France, Nov. 2002.

[7] National Institute of Standards and Technology (NIST). ICAT vulnerability database. `http://icat.nist.gov/icat.cfm`.

[8] OneStat.com. Provider of web analytics. `http://www.onestat.com`.

[9] Protect-data website, http://www.protect-data.com.

[10] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, Dec. 1990.

[11] SecurityFocus. Vulnerability database. `http://securityfocus.com`.

Using Runtime Paths for Macroanalysis

Mike Chen, Emre Kıcıman, Anthony Accardi, Armando Fox, Eric Brewer
UC Berkeley, Stanford University, Tellme Networks
{mikechen, brewer}@cs.berkeley.edu, {emrek, fox}@cs.stanford.edu, anthony@tellme.com

Abstract

We introduce macroanalysis, *an approach used to infer the high-level properties of dynamic, distributed systems, and an indispensable tool when faced with tasks where local context and individual component details are insufficient. We present a new methodology,* runtime path analysis, *where paths are traced through software components and then aggregated to understand global system behavior via statistical inference. Our approach treats components as gray boxes and complements existing microanalysis tools, such as code-level debuggers. We use runtime paths to deduce application state, detect failures, and diagnose problems, all in an application-generic fashion. We have explored path-based macroanalysis both in a research setting and as part of a commercial infrastructure at Tellme Networks.*

1. Introduction

Divide and conquer, layering, and replication are fundamental design principles useful for building large, complex systems, such as Internet services, sensor networks, and peer-to-peer (P2P) systems. Such techniques make building large systems tractable, as they improve availability, increase code reuse, and simplify high-level application structure. Unfortunately, debugging ease and fault monitoring do not scale as well, since global context tends to be dispersed across many small components. Building large, complex systems that are reliable, yet maintainable and extensible, remains a challenge.

Existing debugging techniques make use of various microanalysis tools, such as code-level debuggers and application logs. Such tools tend to provide knowledge limited to component internals, or furnish a thread-level perspective, so that the execution context is lost at the thread boundaries. While they provide valuable, localized knowledge, many of these tools fail to capture aggregate component behavior and macro system properties. By way of analogy, microanalysis allows you to see the details of each honeybee, but macroanalysis is needed to understand how the bees interact to keep a beehive functioning.

Various systems have exposed and exploited non-local system context to address performance and resource allocation problems [1, 15, 17]. Macroanalysis makes use of non-local context to improve system management and reliability. This is especially important for large, dynamic systems, where execution context may be distributed across many components.

One key observation we make about dynamic, distributed systems is that most of them have a single system-wide execution path associated with each request that they service. Examples include Internet services that have request/response paths, and P2P systems and sensor networks that have one-way message paths. By tracing these runtime paths, we expose and connect various local contexts dispersed throughout the system. We then use statistics to analyze many of these paths and thereby better understand the system's behavior.

There are several open, challenging problems that can benefit from the high-level system perspective that macroanalysis provides:

Deducing system structure: Systems evolve through both changes to their components and changes in how these components interact. Understanding such inter-component relationships enables developers and operators to anticipate potential conflicts and debug problems. Unfortunately, current techniques for tracking changes in these relationships rely on error-prone, manual documentation, which is infeasible for rapidly changing systems. As systems grow and increase in complexity, we desire automated mechanisms for deducing system structure and tracking its evolution.

Detecting application-level failures: Despite our best efforts at unit testing and quality assurance, services still fail. Worse still, many application-level faults are only seen by end users after deployment, even though systems are constantly monitored for signs of failure. One large commercial service has found that such errors take considerably longer to detect than lower-level failures. The difficulty is that broken or misconfigured components or bad component interactions may only exhibit symptoms at the application level, and the global context required to programmatically diagnose such application-level failures is usually not available.

Diagnosing failures: Failures often manifest themselves far from their root cause. In the extreme case, faults are not detected within the system's boundaries at all and are only visible to external observers. Unfortunately, existing debugging and diagnosis tools have a limited, local view of the system, and thus work best when failures manifest themselves close to the cause. We desire tools that use global failure information to help operators and developers identify the root cause.

The contributions of this paper are:

1. Recognizing that macroanalysis is critical when developing, evolving, and maintaining reliable systems as they grow in size and complexity.
2. A path-based macroanalysis framework, where we first record the components and resources used to service each request, and then use statistical analysis techniques to deduce system structure, detect application-level failures, and diagnose problems.

We stress that macroanalysis complements, and does not replace, traditional component-oriented systems approaches. We often use such tools to flesh out issues identified via macroanalysis. For example, our failure diagnosis can determine the specific requests and component(s) involved in a failure, but identifying the actual cause may require looking at source code or component logs.

The paper is organized as follows: Section 2 develops the runtime path model. Section 3 describes our analysis framework and current status. Section 4 discusses our results addressing the challenging problems above, both in a research setting and as part of a commercial infrastructure at Tellme Networks. We outline future research directions in Section 5 and discuss related work in Section 6.

2. Runtime Paths

We extend the dataflow paths in Scout [15] and Ninja [18] [1] to incorporate runtime properties. A runtime path is the control flow, resources, and performance characteristics associated with servicing a request. Paths can be recorded during runtime by tracing each request through a live system, spanning the system's layers to access direct component and resource dependencies. Each path then provides a vertical slice of the system from a request's perspective.

We use the term "request" in a broad sense to mean a unit of work. This includes both requests that require responses (e.g., HTTP) and those that don't (e.g., one-way messages).

There are two main requirements for a system to support runtime paths. First, it must be possible to associate a unique path with each distinct request. For example, if the same request is handled by different components in different, possibly distributed processes, we must be able to make this connection, perhaps by using a unique ID that travels with the request. Second, it must be possible to report *observations* and associate them with the components that made them. For a pipelined system, a logging mechanism together with knowledge of component request entry and exit would be sufficient.

[1] Scout: "a logical channel through a multi-layered system over which I/O data flows within a single host". Ninja: "a flow of typed data through multiple services across the wide area".

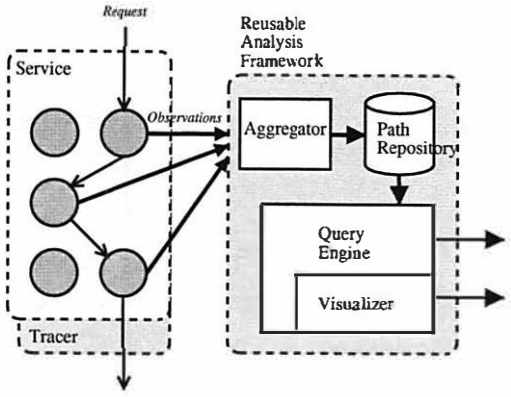


Figure 1. Analysis Framework.

The components make local observations, from which global context is obtained by stringing them together along a runtime path. Each observation contains information about some active component, such as its name, location, timestamp, latency, and arguments. The tracing should be extensible to allow for integration with microanalysis techniques. For example, identifiers could be included in each observation to provide a link with standard application logs.

3. Analysis Framework

The analysis framework consists of five major modules, as illustrated in Figure 1. The *Tracer* tracks requests through the target system, reporting any observations made. Although Tracer is platform-specific, it can be application-generic for platforms that host application components by monitoring requests that enter and exit the components.

The *Aggregator* receives these observations and reconstructs the runtime paths, which the *Path Repository* stores. The *Statistical Declarative Query Engine* handles the data management complexity. It enables us to transparently optimize data storage, and evaluate and update different analysis algorithms. Monitoring and debugging tools should be built on top of this engine. The *Visualization* module helps users understand system behavior. Paths have a natural graph representation: nodes are observations and edges indicate request propagation.

We implemented an extensible Aggregator, a Path Repository, and a Query Engine, with plugins for data clustering and structural anomaly detection. We are currently experimenting with different analysis algorithms.

We built a Tracer for a web server, Jetty, and a clustered J2EE application server, JBoss [10]. The web server inserts a unique request ID into the HTTP request header.

This ID is placed in thread local storage for intra-thread component calls, and passed via a modified RMI library for inter-thread calls. The total code impact was 428 lines in 10 files.

We have been running PetStore, an e-commerce application, and ECperf [20], an industry-standard benchmark for J2EE application servers. Both have a 3-tier architecture consisting of a web server, application components, and a database. The tracing instrumentation is application-generic, so no application changes were necessary.

To measure worst-case performance overhead, we recorded the observations synchronously, using the Java Messaging Service and Java's default serialization methods. We computed a throughput overhead of 16% and an average observation size of 200 bytes – 16 bytes after gzip compression. The compression ratio is high because of redundant text strings, including JVM version identifiers and host names. Using better Java object transport and serialization routines [23] should improve performance.

	Database Tables				
Request Type	Product	Signon	Account	Banner	Inventory
verifysignin	R	R	R		
cart	R			R	R/W
commitorder	R				W
category	R				
search	R			R	
productdetails	R				R/W
newaccount		R	R		
checkout					W

Table 1. An automatically generated partial state dependency table for PetStore. To determine which request types share state, group the rows by common entry under the desired column. For example, the checkout request only writes to the Inventory table, and shares state with three other requests: cart, commitorder, and productdetails.

4. Applying Macroanalysis

4.1. Deducing System Structure

Understanding a service's structure, including the relationship between external requests and the service's internal components and state, enables developers and operators to anticipate potential problems before they upgrade the system. Knowing how components and shared state are used is critical when debugging failed requests. Dependency models have been proposed to improve system reliability and availability [8], but there are few techniques that generate such models automatically.

Key idea: Paths directly capture application structure.

Runtime paths record how a system services real requests, which compares favorably with error-prone, human-generated models and static analysis that predict how the system *might* service such requests. Automatically generated models help developers and operators understand the *actual* behavior of systems under investigation, and can be used as input to recovery mechanisms, such as recursive restarts [4], to reduce mean time to repair.

The Magpie project is using macroanalysis techniques to generate detailed and accurate models of distributed system workloads [14]. While currently used for capacity planning, these models may also be applied to performance debugging, system tuning, and fault diagnosis.

Key idea: Paths associate requests with internal state.

Internet services typically store persistent state in a database to allow for easier front-end scaling. Different requests often share persistent state, but the components handling each request may be unaware of any such sharing. For example, although checkout and login requests may seem like independent HTTP requests, they may share a user profile. A bug during checkout could corrupt this shared state and cause subsequent login failures. Such bugs are difficult to diagnose without an understanding of how various dispersed local contexts depend on each other.

By tracing runtime paths from the web servers, through the application components, and to the databases, we can easily determine how state is shared across requests.

Table 1 shows the mapping between request types and their reads and writes to database tables in PetStore, our desired level of state granularity.

4.2. Detecting Failures using Anomalies

Key idea: Paths often behave differently in failure modes. Hence we can detect failures via changes in path behavior.

Application-level failure detection remains a major challenge today. In practice, quality assurance testing mainly catches simple bugs. Many complex bugs exist in deployed software because of difficulties accurately simulating the workload of a production environment, difficulties modeling the production environment itself, incomplete test coverage, and economic factors. Detecting these bugs in a live system can be difficult, since many bug symptoms are only evident to an end-user, such as incorrect text on a web page.

If we analyze a live system using macroanalysis techniques, however, we can often see secondary effects of failures. Many errors cause runtime paths to end prematurely, while others send paths to less-often used error handlers. Still others, such as fail-stutter faults, simply cause deviations in the latencies of particular path components.

One macroanalysis technique for detecting these changes in path behavior is to search for asymmetries in the interactions among replicated, load-balanced components. Consider an Internet service where all but one of the middle tier nodes are sending queries to a database. Since the middle tier nodes are replicas of one another, it is likely that the

one node's database inactivity is a symptom of some problem. We can extend this idea to treat excessively heavy or light component interaction volume as a sign of failure.

Another technique is to group paths by request type and search for significant deviations in latency or structure.

4.3. Diagnosing Failures through Correlations

Key idea: Runtime paths make the root cause's interaction with a failed request apparent, so that we can quickly explain a failure by a bad component-level or systemic behavior, and can also quickly assess the user visible impact (and hence the priority) of the problem.

When something fails, we want to know why. The challenge here is in tracing externally observed (application-level) failures back to a system fault or root cause.

In practice, we start with sets of suspected failed requests (e.g., reported by users or discovered via anomaly detection) and successful requests, and face the task of identifying the runtime path features underlying any real failures.

The diagnosis task can be cast as a data mining problem, by using data clustering to group the components associated with each failed request's path. Using a previous prototype, Pinpoint [5], we showed that for single component failures the data clustering approach provides a trade-off between accuracy, at 70-90%, and false positives, at 20-40%. This compares favorably with direct fault detection and other automatic analysis methods, which either offer 40% accuracy or many (almost 90%) false positives. Also, we found that paths are vital when dealing with multi-component faults.

Alternatively, the task can be cast as a classification problem in the machine learning domain. Here, the root causes would be the strongest classification rules.

An important benefit of our failure diagnosis approach is that the runtime path data links the logically separate tasks of failure detection and diagnosis, which enables an understanding of mechanisms where the connection between problem causes and symptoms is not otherwise apparent.

4.4. A Commercial Example

Tellme Networks has developed a path-based macro-analysis infrastructure, its *Observation Logs*, to help ensure the high reliability of Tellme's network. Tellme runs voice applications; for the purpose of our current discussion, the system is a telephony network with an Internet back-end, and serviced requests include an audio response to a telephone caller's voice query.

After explaining how runtime paths behave in this example, we will discuss a failure and describe how we used paths to first diagnose the problem, then deduce runtime system structure, and ultimately craft a detection algorithm.

An actual, though simplified, voice response path is illustrated in Figure 2. There are 31 observations in this runtime path, plotted by the relative time at which each observation was made. We call attention to 4 of these, indicating

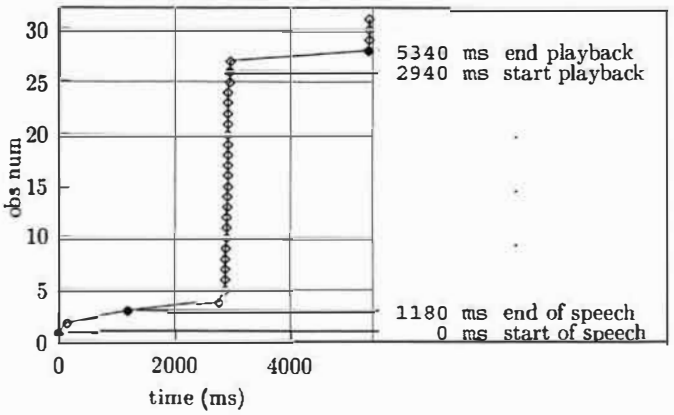


Figure 2. A typical runtime path in Tellme's network.

when the user begins and finishes speaking an utterance and when we start and stop sending audio in response.

The latency profile of the runtime path shown in Figure 2 is close to ideal for this system; most of the time is spent listening to or talking to the user, and the logic connecting the request with the response is rapidly executed. An important metric is the latency between end of speech and start of playback. This 1760 ms delay is perceived by the user, and indicates how quickly the system appears to respond.

Now consider a real network failure. In this case, an isolated process supplying the audio to the machine on the telephony network experienced an internal fault, so that it produced shorter waveforms than the desired ones. This failure is not catastrophic at the system level; a rare, short waveform typically goes unnoticed by the user. The problem is therefore difficult to detect via low-level system monitoring and requires significant application knowledge to handle effectively.

Despite this challenge, we quickly diagnosed this problem using global context in the Observation Logs. An engineer familiar with a particular application noticed a short audio playback and provided details of a phone call that enabled us to quickly locate the relevant runtime path. Once visualized as in Figure 2, a short playback time suggested a truncated waveform. The preceding observations confirmed that a remote process thought it had successfully serviced the audio request, when in fact a rare error had occurred. We identified the particular remote process from the path information, and text logs on that machine subsequently revealed the root cause.

Once we understood the runtime path characteristics for this failure, we were able to query the Observation Logs to detect any similar occurrences throughout Tellme's network. We deduced enough system state to know which components affected which applications, so we could isolate the failing component and assess application impact.

With this new knowledge, we crafted a monitor to use these sub-path latency deviations to detect any future failures in both our production and testing environments.

5. Future Directions

In addition to refining our framework and exploring different algorithms, we are applying our path-based macroanalysis methodology to sensor networks and P2P systems.

Key idea: Violations of macro invariants are signs of system intrusion or buggy implementations. Macroanalysis can help discover invariants, detect violations, and pinpoint the offending components.

Because of the highly distributed and dynamic nature of these systems, many domain-specific macro invariants are difficult to validate using microanalysis or static analysis [7]. Consider an upper bound on the number of hops made during message delivery in a P2P system as a macro invariant. By applying root cause analysis, we can identify peers that incorrectly route messages. In this example, although each node may detect an invariant violation, a diagnosis is difficult without the context contained in a runtime path.

6. Related Work

We consider both microanalysis and macroanalysis work, as well as hybrid approaches.

Macroanalysis Magpie [14] profiles web sites to observe the processing state machine for each HTTP request and to measure request resource consumption (CPU, disk, and network usage) at each stage. The focus is on building probabilistic models of the workload suitable for performance prediction, tuning, and diagnosis.

Microanalysis: Anomaly detection has been used to identify software bugs [7, 22] and to detect intrusions [12], using events based on resource usage [6], system calls [9], and network packets [16]. Paths provide nonlocal context and may make the detection of a new class of intrusions possible.

Hybrid: There are several recent commercial request tracing systems of note. PerformaSure [19] and AppAssure [3] focus on performance diagnosis. IntegriTea [21] focuses on capturing and replaying failure conditions. These systems work with isolated requests, while we aggregate multiple paths and use statistical techniques to infer collective system behavior.

Dynamic program slicing [2] and Whole Program Paths [11] capture dynamic control flow and have been applied to single-process systems analysis.

Some distributed and parallel debuggers support stepping through remote function calls [13]. These tools typically work with individual requests and homogeneous components, and are designed to aid in low-level debugging.

7. Conclusion

Macroanalysis satisfies a need when monitoring and debugging large, complex systems where local context is of insufficient use. To this end, we have presented a runtime path-based approach along with a family of macroanalysis tools that are proving effective in addressing several challenging and important problems. Our method involves dynamically tracing runtime paths through live systems, and recording local observations about performance, interacting components, and resource usage along the way. We subsequently apply data mining techniques to statistically infer aggregate system behavior. This approach is applicable to a large variety of systems, and complements existing microanalysis tools that provide additional insight into individual components.

Our results with distributed Internet systems demonstrate promising progress in 1) deducing system structure and dependencies, 2) detecting failures via path anomalies, and 3) diagnosing problems. We plan to extend our methodology to peer-to-peer systems and sensor networks by validating macro invariants.

References

[1] M. B. Abbott and L. L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.

[3] Alignment Software. AppAssure, 2002. `http://www.alignmentsoftware.com/`.

[4] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *HotOS VIII*, 2001.

[5] M. Chen, E. Kıcıman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Symposium on Dependable Networks and Systems (IPDS Track)*, 2002.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *OSDI*, 2000.

[8] B. Gruschke. A New Approach for Event Correlation based on Dependency Graphs. In *5th Workshop of the OpenView University Association*, 1998.

[9] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[10] JBoss.org. JBoss J2EE Application Server, 2001. `http://www.jboss.org`.

[11] J. R. Larus. Whole Program Paths. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*. Atlanta, GA, May 1999.

[12] W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. In *USENIX Security Symposium*. San Antonio, TX, 1998.

[13] M. S. Meier, K. L. Miller, D. P. Pazel, J. R. Rao, and J. R. Russell. Experiences with Building Distributed Debuggers. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1996.

[14] Microsoft Research. Magpie, 2003. http://research.microsoft.com/projects/magpie/.

[15] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *OSDI*, 1996.

[16] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[17] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual Services: A New Abstraction for Server Consolidation. In *USENIX Annual Technical Conference*, pages 117–130, June 2000.

[18] S. D. Gribble et al. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks*, 35(4):473–497, 2001.

[19] Sitraka. PerformaSure, 2002. http://www.sitraka.com/software/performasure/.

[20] Sun Microsystems. ECperf J2EE benchmark, 2001. http://java.sun.com/j2ee/ecperf/.

[21] TeaLeaf Technology. IntegriTea, 2002. http://www.tealeaf.com/.

[22] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[23] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.

Magpie: online modelling and performance-aware systems

Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan
Microsoft Research Ltd., Cambridge, UK.

Abstract

Understanding the performance of distributed systems requires correlation of thousands of interactions between numerous components — a task best left to a computer. Today's systems provide voluminous traces from each component but do not synthesise the data into concise models of system performance.

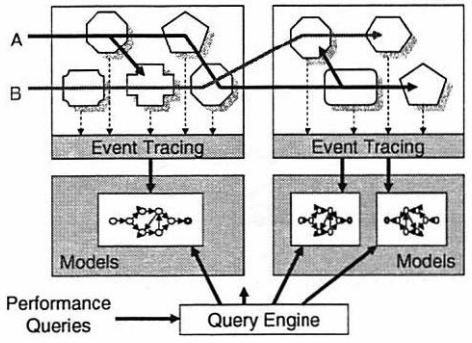
We argue that online performance modelling should be a ubiquitous operating system service and outline several uses including performance debugging, capacity planning, system tuning and anomaly detection. We describe the Magpie modelling service which collates detailed traces from multiple machines in an e-commerce site, extracts request-specific audit trails, and constructs probabilistic models of request behaviour. A feasibility study evaluates the approach using an offline demonstrator. Results show that the approach is promising, but that there are many challenges to building a truly ubiquitious, online modelling infrastructure.

1 Introduction

Computing today is critically dependent on distributed infrastructure. E-mail, file access, and web browsing require the interaction of many machines and software modules. When end-users of such systems experience poor performance it can be extremely difficult to find the cause. Worse, problems are often intermittent or affect only a small subset of users and transactions — the 'it works for me' syndrome.

Aggregate statistics are insufficient to diagnose such problems: the system as a whole might perform quite well, yet individual users see poor performance. Accurate diagnosis requires a detailed audit trail of each request and a model of *normal* request behaviour. Comparing observed behaviour against the model allows identification of anomalous requests and malfunctioning system components.

We believe that providing such models should be a basic operating system service. Performance traces should be routinely collected by all machines at all times and used to generate system performance models which are then made available for online programmatic query. To this end, we are building *Magpie*, an online modelling infrastructure. Magpie is based on two key design principles. *Black-box instrumentation* requires no source code modification to the measured system. *End-to-end tracing* tracks not just aggregate statistics but each individual request's path through the system.



The diagram shows how requests move through different software components across multiple machines in a distributed system. Magpie synthesizes event traces from each machine into models that can be queried programatically.

Figure 1. Magpie architecture.

Fine-grained, low-overhead tracing already exists for Linux [27] and Microsoft's .Net Server [17]; the challenge is to efficiently process this wealth of tracing information for improved system reliability, manageability and performance. Our goal is a system that collects fine-grained traces from all software components; combines these traces across multiple machines; attributes trace events and resource usage to the initiating request; uses machine learning to build a probabilistic model of request behaviour; and compares individual requests against this model to detect anomalies. Figure 1 shows our envisioned high-level design.

In Sections 2 and 3 we describe the many potential uses of online monitoring and modelling. Section 4 describes the current Magpie prototype, which does online monitoring but offline modelling. Sections 5 and 6 briefly describe related work and summarize our position.

2 Scenario: performance debugging

Joe Bloggs logs into his favourite online bookstore, monongahela.com, *to buy 'The Art of Surfing' in preparation for an upcoming conference. Frustratingly, he cannot add this book to his shopping cart, though he can access both the book details and his shopping cart. His complaint to customer support is eventually picked up by Sysadmin Sue. However, Sue is perfectly able to order 'The Art of Surfing' from her machine, and finds nothing suspicious in the system's throughput or availability statistics: she can neither replicate the bug nor identify the faulty component.*

Consider the same website augmented with the Magpie online performance modelling infrastructure. Magpie maintains detailed logs of resource usage and combines them in real time to provide per-request audit trails. It knows the resource consumption of each request in each of several stages — parsing, generation of dynamic content, and database access — and can determine whether and where Joe's request is out of bounds with respect to the model of a correctly behaving request.

Using Magpie's modelling and visualization tools, Sue observes a cluster of similar requests which would not normally be present in the workload model. On closer examination, she sees that these requests spend a suspicious amount of time accessing the 'Book Prices' table, causing a time-out in the front-end server: This is the problem which is affecting Joe and the culprit is a misconfigured SQL Server: Sue could not replicate the bug because her requests were redirected by an IP address based load balancer to a different, correctly configured replica. Within minutes the offending machine is reconfigured and restarted, and Joe can order his book in good time for the conference.

This is just one performance debugging scenario, but there are many others. File access, browsing and e-mail in an Intranet may rely on Active Directory, authentication and DNS servers: slow response times could be caused by any combination of these components. Diagnosing such problems today requires expert manual intervention using tools such as `top`, `traceroute` and `tcpdump`.

3 Applications

Pervasive, online, end-to-end modelling has many uses apart from distributed performance debugging; here we list some of the most exciting ones. These applications are research goals rather than accomplished facts. Section 4 describes our first step towards these goals in the form of our offline modelling prototype.

Capacity planning. Performance prediction tools such as *Indy* [11] require workload models that include a detailed breakdown of resource consumption for each transaction type. Manual creation of such models is time consuming and difficult; Magpie's clustering algorithm (Section 4) automatically creates workload models from live system traces.

Tracking workload level shifts. Workloads can change qualitatively due to subtle changes in user behaviour or client software. For example, a new web interface might provide substring matching in addition to keyword search. To reconfigure the system appropriately, we must distinguish level shifts from the usual fluctuations in workload. Magpie could do so by comparing models of current and past workloads. This might also detect some types of denial-of-service attacks.

Detecting component failure. Failure of software or hardware components can degrade end-to-end performance in non-obvious ways. For example, a failed disk in a RAID array will slow down reads that miss in the buffer cache. By tracking each request through each component, Magpie can pinpoint suspiciously behaving components.

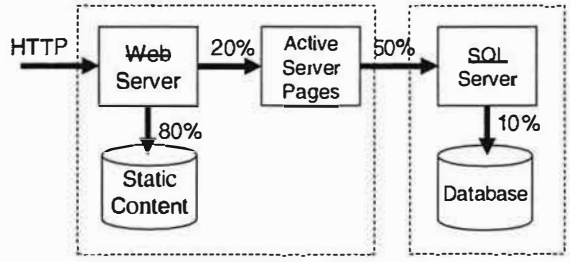
Comparison-based diagnosis. Workload models could be compared across site replicas to diagnose performance discrepancies.

'Bayesian Watchdogs'. Given a probabilistic model of normal request behaviour, we can maintain an estimate of the likelihood of any request as it moves through the system. Requests deemed unlikely can then be escorted into a safe sandboxed area.

Monitoring SLAs. The emerging Web Services standards [26] allow multiple sites to cooperate in servicing a single request. For example, an e-commerce site might access third-party services for authentication, payment, and shipping. Magpie performance models could be compared with service level agreements to check compliance. However, this does raise issues of trust and privacy of sensitive performance data.

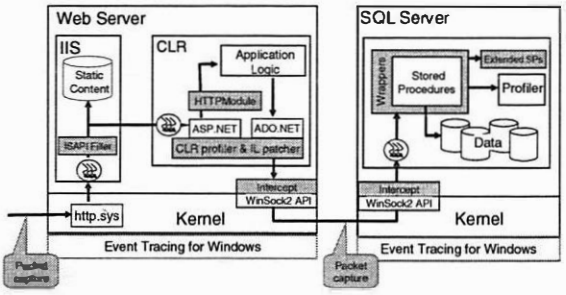
Kernel performance debugging. Response times on a single machine are governed by complex interactions between hardware, device drivers, the I/O manager, the memory system and inter-process communication [14]. Event Tracing for Windows already traces all these OS components, enabling request tracking at this fine granularity.

End-to-end latency tuning. Scalable self-tuning systems such as SEDA [25] optimise individual compo-



In a typical e-commerce site the majority of HTTP Requests are for static content, but a significant fraction require execution of code in the .Net runtime, which might issue SQL queries to a database server machine.

Figure 2. A simple e-commerce site.



Instrumentation points for the web server and database server in our test e-commerce site. Some components such as the http.sys kernel module and the IIS process generate events for request arrival, parsing, etc. Additional instrumentation inserted by Magpie (shown in gray) also generates events; all these events are logged by the Event Tracing for Windows subsystem.

Figure 3. Instrumentation points.

nents for throughput, with potentially deleterious effects on end-to-end latency. For example, they ignore concurrency within a single request, which has little effect on throughput. However, exploiting intra-request concurrency can reduce latency.

4 Feasibility study

To evaluate the feasibility of our approach, we need to answer the following three questions: Are the overheads of tracing acceptable? Is the analysis of log data tractable? Are the resulting models useful?

To this end, we have constructed an *offline* Magpie demonstrator, which traces in-kernel activity, RPCs, system calls, and network communication, using Event Tracing for Windows [17], the .Net Profiling API [18], Detours [12] and `tcpdump` respectively. We then ran an unmodified e-commerce application (Duwamish7) on the two-machine configuration depicted in Figure 2, and exercised it using a workload based on TPC-W [24]. Figure 3 shows the components and instrumentation points for this system.

Each instrumentation point generates a named event, timestamped with the local cycle counter. For example, we have events for context switches and I/O operations as well as entry into and exit from selected procedures. Offline processing assembles logs from multiple machines, associates events with requests, and computes the resource usage of each request between successive events. Figure 4 shows an automatically generated visualization of one such request audit trail.

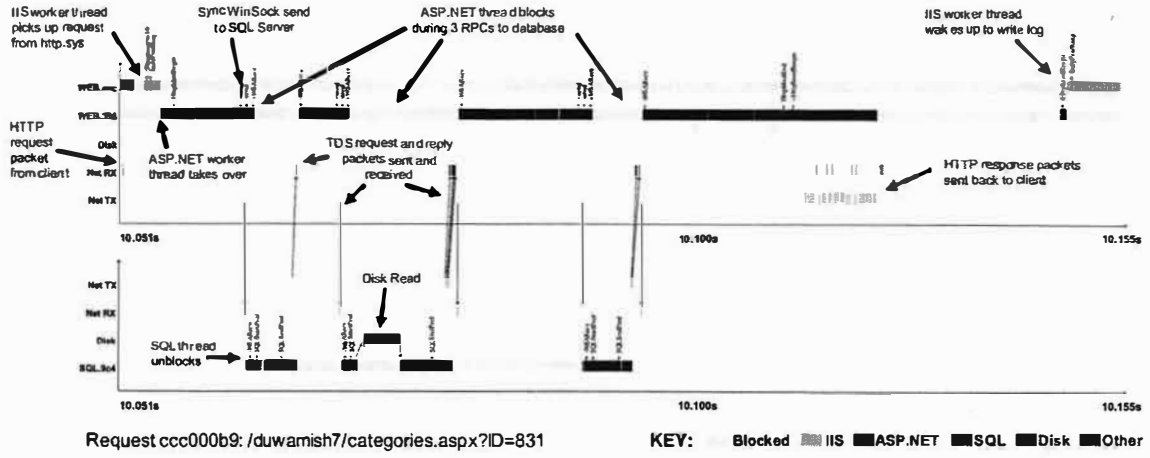
To stitch together a request's control flow across multiple machines, we use the logged network send and receive events. Similarly, we could track requests across multiple thread pools on the web server, by instrumenting the thread synchronization primitives. Our current prototype does not yet track thread synchronization: instead, we track a request across thread pools by adding a unique ID to the request header.

To estimate the worst-case overhead of logging, we ran a simple stress benchmark on our test site, and traced all system activity on both machines. This generated 150k events/min, resulting in 10MB/min of log data. This large volume is mostly caused by inefficient ASCII logging of system calls, and could easily be reduced. On a simple microbenchmark, throughput was reduced by 18%, again with all trace points active.

Behavioural clustering

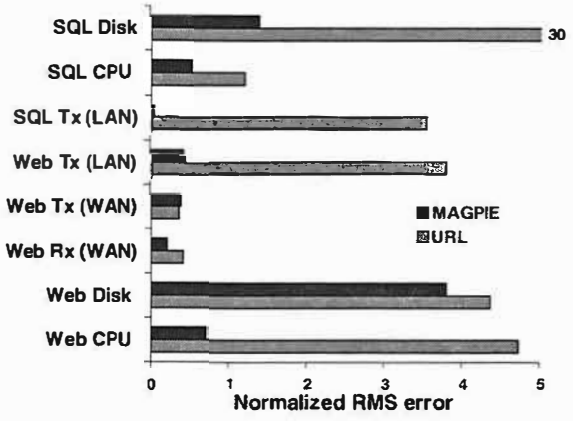
Our initial driving application was workload generation for the Indy performance prediction toolkit. This requires specification of a small number of *representative* transaction types together with their relative frequencies. Typically, these transaction types are categorized by their URL, and measured using microbenchmarks. However, URLs are not always a good indicator of request behaviour: the same URL often takes different paths through the system due to differences in session state or error conditions.

Instead, Magpie categorizes requests by their observed behaviour under realistic workloads. It merges logs and deterministically serializes each request's events to construct an *event string*, annotated with resource usage information. We then cluster the event strings according to the Levenshtein String Edit Distance [21] augmented with the normalized Euclidean distance between the resource usage vectors. Behavioural clustering gives us substantially better clustering accuracy than a URL-based approach, in terms of the difference between behaviour of the representative and those transactions it represents, as shown in Figure 5.



An automatically generated timeline of a sample request across several resources (CPU threads, network, and disk) and different software components on two machines. The small vertical arrows represent events posted by instrumentation points and are labelled with the event name. Annotations explaining how the graph is interpreted have been added manually.

Figure 4. Request audit trail.



Comparison of two clustering methods applied to the same data, one based on Magpie clustering and the other on URL. A synthetic workload is generated using both sets of clusters and compared to the original dataset. This graph shows the RMS error of actual and predicted resource consumption for each model, broken down by resource type (errors are shown normalised by the mean).

Figure 5. Magpie vs. URL-based clusters.

Our unoptimized C♯ implementation takes just under 2 seconds to extract 8 clusters from 5 minutes of trace data (1800 requests) on a 2GHz Intel Pentium 4. This gives us confidence that an online version of Magpie modelling will have low overheads.

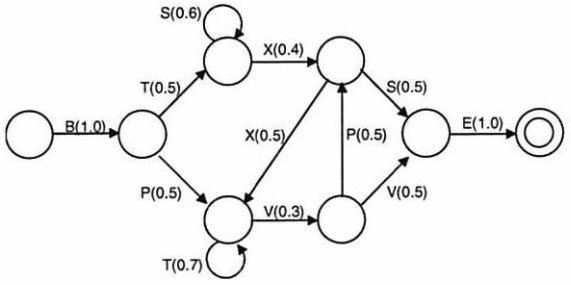
Inferring higher level behaviour

Behavioural clustering alone is sufficient for realistic workload generation, and additionally enables some aspects of the performance debugging scenario of Section 2. By looking for outliers — requests that are not close to any existing clusters — we can identify anomalous requests. However, this does not tell us the cause of the anomaly: the particular event or event sequence that makes the request suspicious.

To identify specific events that are out of place, we must model the process that generates the event sequences. A natural way to model such a process is as a probabilistic state machine: each transition between states produces an event, and has an associated probability. Given such a model and an anomalous request, we can identify events or event sequences with suspiciously low probability. In fact, an online version of such a model could be used to continuously check *all* requests for anomalous behaviour, i.e. to implement the "Bayesian watchdogs" scenario of Section 3.

A probabilistic state machine can be represented as a stochastic context-free grammar (SCFG). This grammar is an intuitive representation of the underlying process which generated the event sequences, and can provide useful clues as to the higher level control flow and the internal structure of the application code, including hierarchy and looping.

Given a set of example strings, the ALERGIA algorithm [4] derives an SCFG in linear time by recursively merging similar portions of their prefix tree. For the test grammar in Figure 6, our C# implementation converges to the correct state machine after 400 sample strings, taking around 30 milliseconds on a 2.5GHz machine. By way of comparison, the example 5 minute long dataset described earlier produced some 1800 event strings, implying that ALERGIA should have a convergence time of about a minute and negligible CPU overhead.



BPTTVPSE	BPVPSE	BPVPSE
BTSSSSSSSSSSXSE	BPTVPXTTVVE	BPTTVVE
BTSXSE	BPVVE	BPVPSE

A stochastic context free grammar (the Reber grammar) and some of the strings which it generates. The string BTSXSE, for example, will be generated with probability 0.06.

Figure 6. Example SCFG.

This efficiency, combined with the enhanced information in the resulting model, has encouraged us to apply ALERGIA to Magpie request event strings. We are currently evaluating ALERGIA's performance on the longer and more complex strings generated by Magpie, and exploring extensions to the algorithm to incorporate resource usage measurements.

Modelling concurrency

Both the clustering and SCFG-based approaches use a deterministic serialization of each request's events, and thus do not model concurrency within a request. In our test scenario there was little intra-request concurrency; more complex systems will require us to explicitly capture concurrency, perhaps by extending the clustering distance metric and replacing SCFGs with coupled hidden Markov models [3]. In so doing, scenarios such as end-to-end latency tuning become feasible.

5 Related work

The closest relative to Magpie is Pinpoint [6], a prototype implementation of the *online system evolution* model [7]. Pinpoint has a similar philosophy and design to Magpie, recommending aggressive logging, analysis and anomaly detection, but its focus is fault detection rather than performance analysis. TIPME [8] performs continuous monitoring on a single machine, also for the purposes of debugging particular problems. In this case, the monitored environment encompasses the transactions initiated by user input to the X Windows system on a single machine.

Scout [19] and SEDA [25] require explicitly defined paths along which requests travel through the system. In contrast, Magpie infers paths by combining event logs generated by black-box instrumentation. Whole Path Profiling [16] traces program execution at the basic block level; Magpie's paths are at a much coarser granularity, but can span multiple machines.

Log-based performance profiling has been used in distributed systems [13], operating systems [23], and adaptive applications [20]. Magpie differs in that its logging is black-box and not confined to a single system. It also tracks resource usage of individual requests rather than aggregating information to a system component or resource.

Distributed event-based monitors and debuggers [1, 2, 15] track event sequences across machines, but do not monitor resource usage, which is essential for performance analysis. Systems such as that used by Appliant [5] measure web application response time by embedding JavaScript in the HTML of fetched pages which records the relevant data at the client browser. The aggregated data gives a view of server latency which would complement the detailed server-side workload characterisation obtained using Magpie.

Finally, a few model checking approaches infer correctness models from source code analysis [9] or runtime monitoring [10, 22]; this is similar to our approach of inferring performance models.

6 Conclusion

Our preliminary results show that fine-grained logging and offline analysis are feasible, and that the resulting models are useful for workload generation. Truly pervasive, online modelling has many potential applications, but also presents many challenges.

Not all events have information value for all applications. For example, we both instrument TCP sends and receives, *and* capture packets on the wire, redundant except when debugging TCP. Ideally, we would dynamically insert and remove instrumentation according to its utility.

Our modelling algorithms are offline, operating on a single merged log of all events. Online modelling will require incremental, distributed algorithms that can process events when and where they occur. We are currently developing the infrastructure to enable an online system. In addition, we need efficient ways to use the generated models: for example, a distributed database with an online query mechanism for performance debugging.

Although some of our target scenarios are ambitious given the current state-of-the-art, we believe that they will be achievable in the near future. Performance-aware systems are an important step towards automatic system management and an essential part of managing increasingly complex and distributed systems.

References

[1] *E. Al-Shaer, H. Abdel-Wahab, and K. Maly.* HiFi: A new monitoring architecture for distributed systems management. *Proc. IEEE 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 171–178, May 1999.

[2] *P. C. Bates.* Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems (TOCS)*, 13(1):1–31, 1995.

[3] *M. Brand, N. Oliver, and A. Pentland.* Coupled hidden Markov models for complex action recognition. *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR'97)*, pages 994–999, June 1997.

[4] *R. Carrasco and J. Oncina.* Learning stochastic regular grammars by means of a state merging method. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence*, pages 139–152. Springer-Verlag, 1994.

[5] *J. B. Chen and M. Perkowitz.* Using end-user latency to manage internet infrastructure. *Proc. 2nd Workshop on Industrial Experiences with Systems Software WIESS'02*, Dec. 2002.

[6] *M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox.* Pinpoint: Problem determination in large, dynamic, Internet services. *Proc. International Conference on Dependable Systems and Networks (IPDS Track)*, pages 595–604, June 2002.

[7] *M. Y. Chen, E. Kiciman, and E. Brewer.* An online evolutionary approach to developing Internet services. *Proc. 10th SIGOPS European Workshop*, Sept. 2002.

[8] *Y. Endo and M. Seltzer.* Improving interactive performance using TIPME. *Proc. ACM SIGMETRICS*, June 2000.

[9] *D. Engler, D. Y. Chen, and A. Chou.* Bugs as inconsistent behavior: A general approach to inferring errors in systems code. *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 57–72, Oct. 2001.

[10] *M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin.* Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[11] *J. C. Hardwick.* Modeling the performance of e-commerce sites. *Journal of Computer Resource Management*, 105:3–12, 2002.

[12] *G. C. Hunt and D. Brubacher.* Detours: Binary interception of Win32 functions. *Proc. 3rd USENIX Windows NT Symposium*, pages 135–144, July 1999.

[13] *G. C. Hunt and M. L. Scott.* The Coign automatic distributed partitioning system. *Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 187–200, Feb. 1999.

[14] *M. B. Jones and J. Regehr.* The problems you're having may not be the problems you think you're having: Results from a latency study of Windows NT. *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Mar. 1999.

[15] *J. Joyce, G. Lomow, K. Slind, and B. Unger.* Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 5(2):121–150, 1987.

[16] *J. R. Larus.* Whole program paths. *Proc. ACM conference on Programming language design and implementation (SIGPLAN'99)*, pages 259–269, June 1999.

[17] *Microsoft Corp.* Event Tracing for Windows (ETW). `http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp`, 2002.

[18] *Microsoft Corp.* .Net Profiling API. `http://msdn.microsoft.com/library/en-us/cpguide/html/cpcondebuggingprofiling.asp`. 2002.

[19] *D. Mosberger and L. L. Peterson.* Making paths explicit in the Scout operating system. *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 153–168, Oct. 1996.

[20] *D. Narayanan, J. Flinn, and M. Satyanarayanan.* Using history to improve mobile application adaptation. *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications*, pages 31–40, Dec. 2000.

[21] D. Sankoff and J. Kruskal, editors. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison*, chapter 1. CLSI Publications, 1999.

[22] *S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson.* Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[23] *M. Seltzer and C. Small.* Self-monitoring and self-adapting operating systems. *Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 124–129, May 1997.

[24] Transaction Processing Performance Council. *TPC Benchmark W (Web Commerce) Specification*. `http://www.tpc.org/tpcw/`.

[25] *M. Welsh, D. Culler, and E. Brewer.* SEDA: An architecture for well-conditioned, scalable Internet services. *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 230–243, Oct. 2001.

[26] *World Wide Web Consortium.* Web Services. `http://www.w3c.org/2002/ws/`. 2002.

[27] *K. Yaghmour and M. R. Dagenais.* Measuring and characterizing system behavior using kernel-level event logging. *Proc. USENIX Annual Technical Conference*, June 2000.

Using Computers to Diagnose Computer Problems

Joshua A. Redstone, Michael M. Swift, Brian N. Bershad
Department of Computer Science and Engineering
University of Washington, Seattle
(redstone, mikesw, bershad)@cs.washington.edu

Abstract

Although computers continue to improve in speed and functionality, they remain difficult to use. Problems frequently occur, and it is hard to find fixes or workarounds. This paper argues for the importance and feasibility of building a global-scale automated problem diagnosis system that captures the natural, although labor intensive, workflow of system diagnosis and repair. The system collects problem symptoms from users' desktops, rather than requiring users to describe their problems to primitive search engines, automatically searches global databases of problem symptoms and fixes, and also allows ordinary users to contribute accurate problem reports in a structured manner.

1 Introduction

Despite continuous advances in hardware and software technology, computers are still difficult to use. They often behave in unexpected ways, and it is hard to find fixes or workarounds for problems encountered. The typical approach to solving a problem is to describe the symptoms (e.g. "Word footnotes don't work") to the keyword search interface of a vendor-owned help database, a small number of public databases, and then finally a broad "Google"-like search of the entire web. With luck, the "right" choice of keywords may quickly produce an article or posting describing the problem, the cause, and hopefully a resolution. More likely, though, the user gets back too little, too much, or the wrong information. He may continue searching, contact customer support or a message board, or simply give up and hope the problem doesn't come up again. This can be time consuming and frustrating. Moreover, for a given problem, this diagnostic process is repeated for each user touched by the problem, leading to massive global costs as a single problem is diagnosed millions of times. In contrast, root cause analysis and repair is done relatively infrequently. Once a user or company has identified a problem's symptoms and repair procedures, he or it posts a solution to some database, with the intent of sharing the solution with everyone. Unfortunately, the high global cost of finding the solution substantially reduces the value of having it in the first place.

Today, computers scan genetic sequences to identify the root causes of disease, pinpoint DDOS attacks on the Internet, and even match up lonely singles based on personality profiles. Yet, they are nearly useless when our computers don't do what we expect, even when the same problem has occurred a thousand times before on a thousand different machines.

This paper presents the simple vision in which computers diagnose their own problems, leveraging prior analysis work done by others. In line with this vision, we propose that problem reports, which today are unstructured text, follow a structured format and in particular that they express symptoms and causes in a machine-readable and machine-testable format. A structured representation simplifies diagnosis since an errant client machine can search for and test itself against symptoms in a global database with high precision. The structured representation, does, however, complicate the task of problem reporting. While true, we believe that finding and fixing a problem for the first time is where the hard work occurs, and that any incremental burden posed by representing that process in a structured format is small.

1.1 Why now and not before?

In the history of computing, there's never been a time that the system, and not the user, has been responsible for closing the gap between system behavior and user expectations. Why, then, is *now* the right time to start building automated, global-scale, diagnostic services? Consider these enabling factors:

- First, the Internet generates the ultimate network effect, making it possible for anyone, anywhere on the planet, to derive value from the prior experiences (good and bad) of others. Proxy caches, peer-to-peer file sharing networks and even user-contributed product reviews have shown this.

- Second, although there are hundreds of millions of machines, there is relatively little variety in hardware and software. Consequently, a problem found and fixed on one machine is likely to be found and fixed in the same way on another.
- Third, many operating systems support a generalized form of configuration management, such as the Windows registry. This makes it feasible to automatically determine the configuration of a machine.
- Fourth, standardized user-interfaces facilitate the mechanical recording of user-interface events and hence discovery of symptoms.
- Fifth and finally, vendors are making more of their bug databases public (their paranoia is mitigated by the economic benefit of avoiding direct contact with the customer), so there exist large, high-quality sources of known problems that can be automatically diagnosed.

Because of these reasons, an effective diagnosis solution can be built today, using existing operating systems and applications.

However, as we will discuss, there is an opportunity to build an even more effective solution by extending today's operating systems and applications so that a diagnosis engine can observe all state and behavior of the computer, across all applications. At first glance, it may seem as though any additional operating system or application work to support automated diagnosis creates a new burden, and therefore development cost, to be borne by software manufacturers. However, we argue that automated diagnosis is actually a necessary component of any system claiming *high availability* as a feature. Although availability is typically measured in terms of uptime – how long since the last crash – this system-oriented perspective is irrelevant to the user. Instead, a user perceives availability in terms of *goodtime*, which is uptime less the amount of time spent figuring out why the system isn't doing what the user expects. As an example, one of the authors of this paper recently switched his day-to-day working platform from Windows XP to another operating system. Although Windows XP uptime was much improved over its predecessors, goodtime was not. In contrast, uptimes on this other operating system are roughly the same as with Windows XP, but the goodtimes are better. While we don't intend to justify our position with a single anecdote, it should be clear that the broad platform coverage of most applications is becoming such that users can easily migrate to those systems where goodtimes are plentiful.

The goal of this paper is to encourage the systems community to take seriously the challenge of automatically diagnosing system and application problems, and to show that there exists a reasonable path that gets us from here to there. To be clear, it is a path that we ourselves have not traveled as we have not built the system we describe. Consequently, our assumptions are unchecked, and there may be some very good reasons why the state of the art in systems diagnostics is and will remain a glorified version of *grep*. But, it seems unlikely.

In the next section, we present additional background material so that the reader can differentiate from what has been and what remains to be done. In Section 3 we sketch out one possible solution to the problem of automated diagnostics. Finally, in Section 4 we conclude.

2 Background

Software vendors, in order to reduce customer support costs, are, and have been, motivated to provide some sort of diagnostic function with their systems. Most primitively, one can even find occasional bug reports at the bottom of some thirty-year-old UNIX man pages. Where the software vendors have left off, user communities have picked up with their own FAQs and bulletin boards. Indeed, the theme of "why bad things happen to good computers" has spawned an entire book genre dedicated to more goodtime. Below, we briefly characterize a few existing solutions according to whether they are manual or automatic.

2.1 Manual diagnosis

Manual problem diagnosis has the user searching public information sources for a problem report. There are two common information sources: vendor-controlled databases (such as [1] or [11]) and community databases (such as [3] or [16]). Vender-controlled databases have the advantage of providing high-quality coverage of a specific class of problems. However, they are limited in scope and are closed – contribution is tightly controlled. Limiting contribution means that database information may be stale and may contain omissions due to broader corporate considerations.

Community databases, such as discussion boards and mailing lists, offer more wide-ranging and up-to-date information. However, with no standard format for articles, locating information is especially difficult. In addition, information may be inaccurate because there is no quality control mechanism. Posting to these forums can be effective, but often requires an extended dialog to describe key symptoms and configuration details.

Searching is difficult in both kinds of databases because the search interface is inefficient and error prone. Most systems offer only keyword search, although a few natural language systems exist [7]. Successful keyword search requires choosing the correct terminology, which in turn frequently requires

detailed technical understanding of the problem. Further, that terminology may not be the same for all databases. Natural language systems promise to improve search quality, but still require sufficiently detailed understanding of the problem to formulate a specific request.

Even when a user's search locates a possible cause report, the user must manually determine if the system diagnosed in the report is "phylogenetically similar" to the system in question. Often, this is impossible, as the information in the report insufficiently describes the elements of the reported system. At other times, the system may be well-described, leaving it to the user to determine the differences between the systems, and then if they matter (e.g., difference in BIOS versions). For example, a printer may not print because a user has the wrong driver for the printer, or because the driver is installed in the wrong directory, with each possible cause described in a separate report. A user who has the wrong driver installed in the wrong directory will unprofitably apply the fix from one report without considering the second.

2.2 *Automated diagnosis*

An automated diagnosis service shields the user from the details of determining the source of a problem, and focuses instead on revealing the solution. Broadly, there are two types of problems dealt with in automated systems:

Type I: These problems are those for which the resolution is to change the system (upgrading an application, fixing the registry, etc.), and typically result from some well, or partially, understood bug.

Type II: These problems are those for which the resolution is to change user behavior (e.g., saving as a different file format). These problems are either due to correct but undesired system behavior or to a bug for which a fix is not known.

Addressing Type I problems are systems such as WindowsUpdate [14], Windows Baseline Security Analyzer [10] and virus scanners [5], which scan a computer and list available software updates or fixes. However, these solutions lack the ability to diagnose specific problems (they find all bugs with known fixes rather than the bug you want fixed), and the first two may introduce new problems by fixing bugs that are not experienced. By analogy, one can imagine a general practitioner who prescribes a lung transplant in order to cure a patient's nagging cough.

The Windows Error Reporting System ([12] and [13]) suggests fixes for Type I problems, but only those that cause crashes, which occur much less often than general usability problems. Upon a crash, it alerts Microsoft, reporting the loaded executables and their versions. In some cases, Microsoft is able to identify the bug and provide a fix immediately. Autonomic Computing [2][9] also addresses only Type I problems by monitoring system behavior, and then tuning or repairing the system as appropriate. The operative analogy with these systems is the mechanic who replaces your car's brakes after you've run into a wall because you were distracted trying to figure out how to turn off the windshield wipers. The problem is with the wipers, not the brakes.

Agent based approaches such as the Microsoft Office Assistant [8] target the subset of Type 2 problems that do not include bugs. Agents passively monitor user activity and actively offer suggestions based on observed behavior. Such systems are a step in the right direction, but we believe that greater depth, coverage, and specificity are required.

To summarize, existing manual and automated tools provide only limited assistance in diagnosis. Manual tools can diagnosis a wide range of problems but imprecisely and at high cost. Automated tools provide low-cost assistance, but only cover a limited range of problems and do not provide targeted assistance for particular problems.

3 Automated Problem Diagnosis

An automatic problem diagnosis system has three components. The first component captures aspects of the computer's state and behavior necessary to characterize the problem. This includes the symptoms and information such as the installed applications and their versions. The second component matches this information against problem reports to identify the report(s) that best diagnose the problem. Finally, the third component produces new problem reports as new problems are discovered. This section discusses issues in the design of each component.

3.1 *Observing Symptoms*

The flexibility of a diagnosis system to handle different problems depends on its ability to observe the problem. This is a challenging task. To collect the information necessary to perform a diagnosis, all relevant information must be visible to the diagnosis engine. For example, diagnosing a bug in which a missing library causes an application to display an error message when launched involves observing the error message and that the library is missing. If the tool cannot observe the error message or detect the library's existence, it cannot diagnosis this particular problem. In addition, state and behavior should be observable at the correct level of abstraction for robust diagnosis. For example, observing that the 'OK' button was clicked may be more useful than observing that a mouse click occurred at pixel (x,y). Translation between abstraction levels is possible, but may be difficult to accomplish in a robust manner.

Computers today already expose a wide variety of state and behavioral information, allowing for the diagnosis of a wide range of problems. However, gaps exist, and ultimately, robust diagnosis will require changing systems and applications to reveal more information. The following paragraphs discuss the information available today. We divide the discussion of observable state and behavior information into three categories depending on the agent: the user, applications, or the operating system.

User behavior is revealed in terms of the user's input to the system. Since input is an OS service, it is readily available for observation. Most systems allow capture of mouse and keyboard events. Many applications use standard user-interface toolkits, which interact with system resources in an easily observable way. For these applications, events such as menu and dialog box activity are visible. However, observing user behavior in applications that manage their own user interface requires their cooperation in some form.

Applications serendipitously expose a fair amount of behavior and state by virtue of their interaction with the operating system (and thus, hardware). For example, it is straightforward to capture calls to system APIs, including registry (on Windows) and file system accesses. However, applications will not reveal much internal information without modification. Many applications support a debug interface, usually used during development, or export an API for extensibility. Exploiting this functionality may offer a low cost path to accessing more internal application state and behavior.

The operating system exposes many aspects of state and behavior already for the purpose of system management, through performance monitors and event logs. Furthermore, useful configuration information is available from the file system and registry (in Windows) or /etc (in *nix). In addition, the OS exports a rich programming interface for discovering and manipulating system state. Therefore, tools can adequately capture OS behavior and configuration today.

Diagnosis fundamentally involves gathering information, which raises privacy concerns. Users today explicitly choose which information to share with support forums or help desks, and the diagnosis system should provide similar options. If the diagnosis process involves an untrusted computer, then system state must be filtered to avoid disclosing sensitive data. Or, the diagnosis process can be performed only on trusted computers (e.g., by downloading problem databases.)

Examples

We briefly give three examples to show how an automated diagnosis system might detect that a user is having problems. The first example shows what can be done using today's system and application infrastructure. The second illustrates why changes to infrastructure may be required. Finally, the third problem is representative of a class of problems which we believe are not detectable using automatic means.

Problem 1: Quicktime is not installed. When a user clicks on a Quicktime URL, she expects a movie to play. Using today's system and application interfaces, a diagnostic service could observe the HTTP request for a URL ending in .mov, and a subsequent dialog box containing the message that there is no application to display this object. This is sufficient to scan the problem reports database and determine that Quicktime needs to be installed.

Problem 2: Page doesn't render properly. A more difficult problem might be that an HTML page won't render properly. Suppose that the particular cause is that too many display elements exceeds the browser's internal limits. For a diagnosis tool to recognize this, the browser would have to export sufficient information on its internal state and dynamic document contents.

Problem 3. Page doesn't print properly. Finally, diagnosing some problems, such as those that reveal themselves externally to the system, will be difficult without significant additional infrastructure. Consider a bug in which a document prints incorrectly. The information necessary to diagnose the problem, namely the printed document, is not visible to the computer. Even if the computer could observe the document, it may be challenging to characterize in a general way the qualities of the document that make it unsatisfactory. It may be necessary to involve the user here.

3.2 A Structured Database

The second element in a diagnosis system is a mechanism to search problem reports for those that match the observed state and behavior. We believe the way to accomplish this is to require that problem reports be written in a structured, machine-readable format, such as XML. XML provides a semi-structured format for reports with variable levels of detail, and a rich query language for matching symptoms to reports [4]. Future advances in natural language techniques may allow more flexible problem report formats. Automated capture of symptoms for both searching and reporting greatly increases the accuracy of searches compared to today's manual searches.

Searching the database, at the conceptual level, consists of comparing each problem report to the state and symptoms of the system. The reports can then be ranked according to how well they match.

3.3 *Generating New Reports*

The third and final element of the system generates new problem reports. While automated diagnosis is useful even if it can only recognize a few problems, the technique works best when leveraging the experiences of all who experience problems. The key is to make problem reporting as simple as possible. Manual entry of reports is always an option, but we believe that the technology of the other diagnosis components can aid this process. For example, when a user attempts to diagnose a problem and no problem report is found, the state and behavior information collected to aid in diagnosis can be used to construct a new problem report. There are similar privacy concerns when generating new reports as when collecting system state for diagnosis. The ultimate goal is to construct new reports without any user intervention at all. To distinguish the quality of reports, since they come from many sources, a community rating system, such as used at eBay.com [6], can be used.

3.4 *Summary*

Each of the three components of problem diagnosis described here has been inspired by counterpart work in other fields, which provides comfort that the ultimate solution is achievable. The TIGR [18] and NCBI [15] genome database projects aggregate gene data from many researchers, and leverage a standard data format to allow users to easily benefit from every researcher's contributions. The SDSS SkyView astronomy database [17] provides advanced online query access to astronomical data. The intelligent query interface provides efficient access to a large amount of data. eBay.com [6] has a feedback system to judge the quality of data from uncontrolled sources. Finally, the Windows Error Reporting Service [12] relies on technology to automatically construct new problem reports when users experience a crash without input from the user.

4 Conclusion

Despite continuous advances in hardware and software, computers are still difficult to use and users frequently have problems. Computer usability is essentially an availability issue: if the user can't get their job done, it is irrelevant that the computer is running. The correct metric of computer availability therefore is not the traditional *uptime* metric, but instead is the user's ability to get work done (*goodtime*).

This paper argues that an important way to increase *goodtime* is to decrease the time spent diagnosing problems by automating the diagnosis process. When the user encounters a problem, the computer should examine the state and behavior of the machine, search problem databases for a matching problem report, and present the diagnosis to the user. Ideally, this process should occur without any interaction from the user.

We believe that automating problem diagnosis is possible for a large class of problems with today's operating systems and applications, and can only improve with further operating system and application support. The challenges in building an automated diagnosis system are capturing relevant state and behavior, matching to problem reports, and creating new problem reports as new problems arise. Capturing state and behavior is possible because critical systems and applications already have interfaces and logs for determining the state and activities of a system. However, some problems are undetectable with the information available from applications and the operating system today. To perform robust detection, we need new interfaces in systems to allow applications to expose their internal state and behavior. Performing matching automatically is possible if we structure problem reports in a machine-readable format.

The state of system and application support for automated problem diagnosis today is not unlike the state of support for system auditing and event recording in early operating systems which were extraordinarily difficult to monitor. Then, to enable even the simplest of management functionality, applications and the operating had to change to expose their behavior to monitoring services via mechanisms such as SNMP. Similarly, automated problem diagnosis also requires application and OS modification. As with those earlier manageability overhauls, we believe that small investments in the way we build applications and operating systems will yield big returns.

5 References

[1] Apple Corporation. *AppleCare Knowledge Base*, http://kbase.info.apple.com.

[2] G. Banga. *Auto-diagnosis of Field Problems in an Appliance Operating System*, in Proceedings of the USENIX Technical Conference, June 2000.

[3] BugNet. *BugNet*, http://www.bugnet.com.

[4] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Sim6on and M. Stefanescu. *XQuery 1.0: An XML Query Language*, W3C Consortium, http://www.w3.org/TR/xquery.

[5] D. Chess. *Virus Verification and Removal Tools and Techniques*, Virus Bulletin, November 1991.

[6] EBay Corporation. *Feedback Profiles*, http://www.ebay.com.

[7] D. Heckerman and E. Horvitz. *Inferring Informational Goals from Free-Text Queries: A Bayesian Approach*, in Proceedings of the

Fourteenth Conference on Uncertainty in AI, July 1998.

[8] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. *The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users*, in Proceedings of the Fourteenth Conference on Uncertainty in AI, July 1998.

[9] IBM Corporation. *Autonomic Computing*, http://www.research.ibm.com/autonomic.

[10] Microsoft Corporation. *Microsoft Baseline Security Analyzer*, http://www.microsoft.com/technet/security/tools/Tools/MBSAhome.asp.

[11] Microsoft Corporation. *Microsoft Knowledge Base*, http://support.microsoft.com.

[12] Microsoft Corporation. *Windows Error Reporting*, http://msdn.microsoft.com/library/en-us/debug/base/windows_error_reporting.asp.

[13] Microsoft Corporation. *Windows Online Crash Analysis*, http://oca.microsoft.com/en/Welcome.asp.

[14] Microsoft Corporation. *Windows Update*, http://www.windowsupdate.com.

[15] J. Ostell and J. Kans. *The NCBI data model.* Methods of Biochemical Analysis, Vol. 39, July 1998.

[16] Redhat Corporation. *Redhat Support Forums*, http://www.redhat.com/support/forums.

[17] A. Szalay, J. Gray, A. Thakar, P. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. *The SDSS SkyServer – Public Access to the Sloan Digital Sky Server Data*, in Proceedings of ACM SIGMOD, June 2002.

[18] TIGR. *The Institute for Genomic Research.* http://www.tigr.org.

Using Performance Reflection in Systems Software

Robert Fowler†, Alan Cox†, Sameh Elnikety‡, and Willy Zwaenepoel‡
† Department of Computer Science, Rice University, Houston, Texas, USA.
‡ School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland

Abstract

We argue that systems software can exploit hardware instrumentation mechanisms, such as performance monitoring counters in modern processors, along with general system statistics to reactively modify its behavior to achieve better performance. In this paper we outline our approach of using these instrumentation mechanisms to estimate *productivity* and *overhead* metrics while running user applications. At the kernel level, we speculate that the scheduler can exploit these metrics to improve system performance. At the application level, we show that applications can use these metrics as well as application-specific productivity metrics to reactively tune their performance. We give several examples of using reflection at the kernel level (e.g., scheduling to improve memory hierarchy performance) and at the application level (e.g., server throttling).

1 Introduction

Traditional processors have real-time clocks and interval timers. Today cycle counters and event counters are universally available in modern processors and chipsets. For example, the AMD Athlon processor has four performance monitoring counters (PMC) that can be programmed to count specific performance-related events such as TLB and cache misses. Moreover going beyond simple counters, the Alpha EV67 processor and its successors includes a ProfileMe [5] facility that can record the execution history of a single instruction as it passes through the pipeline. We expect that future processors and chipsets will have even more hardware instrumentation mechanisms.

The most common use of these mechanisms is by programmers to do performance debugging for application code. Also, these mechanisms have been successfully applied to the analysis and tuning of application code through compiler [8] and link-time optimizations, and applied to architecture evaluation. Despite the long history of using hardware instrumentation mechanisms, few studies focused on using them to reactively change kernel and application behavior.

We advocate using currently-existing hardware instrumentation mechanisms as the foundation of a *kernel performance reflection* facility designed to collect real time performance information to reactively modify operating system and application behavior.

The rest of the paper is structured as follows: Section 2 describes our approach of using performance reflection. We discuss examples of using performance reflection in OS kernels and in applications in sections 3 and 4, respectively. Section 5 presents related work. Finally, we present our conclusions in section 6.

2 Our Approach

We propose adding a performance reflection facility to the OS kernel to collect performance metrics using timers, event counters, and some programmed hooks. These metrics can estimate *overhead* and *productivity*.

First, some metrics represent costs: For example, the TLB and data cache miss rates measure the overhead that the system incurs in running the applications. We use these metrics to estimate *overhead*.

Second, some metrics count useful work: For example, the number of instructions executed, the floating point operation (FLOP) rate, bytes transferred

to I/O devices, and the percentage of time the CPU spends in user mode are measures of useful work done. We use these metrics to estimate *productivity*.

We use the relationship between overhead and productivity to determine if there is a need to tune the system. Figure 1 shows three schematic plots that represent different relationships between overhead and productivity. In the first plot, both overhead and productivity are increasing, indicating that the load on the system is increasing and the system is behaving well. The second plot shows that the productivity is decreasing whereas the overhead is increasing. This corresponds to an undesired condition, such as thrashing when the system is in overload. Finally in the third plot, both overhead and productivity are decreasing, indicating a normal behavior as the system load decreases.

There are different ways of estimating productivity and overhead. It is not necessary to use any specific metric for these estimates. It is also possible to compute some metrics indirectly [1] if they are not available from the hardware. For example, the Cycles Per Instruction (CPI), which is a common measure of processor productivity, can be computed over some interval by taking the ratio of the number of cycles to the number of instructions graduated.

While many different kinds of events can be counted, the number of distinct measures of productivity counted either by hardware counters or the OS is small, perhaps including instructions, FLOPs, and bytes/packets transferred over I/O devices. Similarly, a small set of cost/overhead measures (cycles, L2 cache misses, TLB misses, interrupts) is sufficient. The kernel can use its own heuristics, or it can be guided by application advice provided through an interface similar to `madvise(3)`.

Productivity estimates can be enhanced with application cooperation. A variable shared between the application and kernel can be used by the application to inform the kernel of its rate of progress [6]. For example, a multi-threaded network server, such as a Web server or a file server, can use the number of requests served as its measure of progress and requests per unit time as its productivity metric.

3 Use of Performance Reflection in OS Kernels

The OS kernel can use the overhead and productivity metrics to reactively change its policies, such as changing the quantum size or the scheduling policy. In this section, we discuss a few applications of performance reflection in scheduling.

3.1 Memory Hierarchy Performance

Memory hierarchy performance can be very sensitive to competition on shared resources. For example, the standard configuration of IBM Regatta node has modules containing two Power4 processors that share a common cache and interface to main memory. Since it is known that many large scientific programs are memory-bandwidth bound, there is also an HPC variant of the hardware that contains only a single processor per module. For bandwidth-limited applications the second processor adds little or no additional performance and eliminating it saves cost while further eliminating possible cache interference. While it would not save the cost of the extra processors, monitoring miss rates of the shared cache of a standard node would enable the system to either schedule only one thread per module or to possibly identify "compatible" threads to co-schedule.

Similar scheduling strategies [9, 11] have been proposed for use with Simultaneous Multi-threading (SMT) [13].

The performance of non-uniform memory access (NUMA) machines is dependent on the assignment of threads to processors. The kernel can monitor memory behavior by, depending on the level of architectural support, measuring remote references, cache miss behavior, or cycles per instruction (CPI). Thread rescheduling decisions can then be based on this feedback.

4 Use of Performance Reflection in Applications

Applications can use the overhead and productivity metrics provided by the kernel, as well as

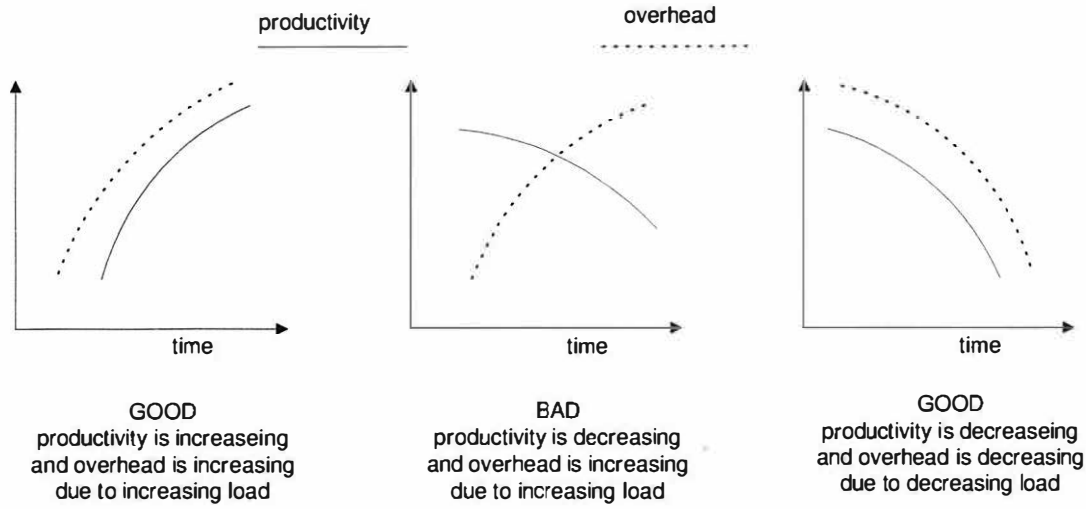


Figure 1: The relationship between overhead and productivity.

application-specific productivity metrics to reactively change their policies.

4.1 Server throttling

To demonstrate the feasibility of using reflection in applications, we show that our techniques can be used to do server throttling. Our experimental setup contains a MySQL database server running under Red Hat Linux 7.0 with the 2.4.18 kernel on an AMD Athlon 1.3 GHz processor. We used the shopping mix of the TPC-W [12] workload to drive the database server. Under this workload, the database server is the bottleneck. The database server thrashes when the number of concurrent queries is too high.

We developed a simple controller that uses the overhead and productivity metrics to dynamically determine the concurrency level of the database by controlling the number of active database connections. The controller receives all requests for database connections. It queues excess requests if the demand exceeds the number of available connections. When connections are released or more connections become available, queued requests are satisfied. The controller uses the PerfCtr [10] kernel module to read the number of L1 DTLB misses, L2 DTLB misses, and L1 and L2 data cache misses from the Athlon AMD processor [4].

The controller uses feedback from the kernel including the DTLB and data cache miss rates to estimate the overhead metric. It uses the percentage of user-mode CPU utilization and the throughput rate to estimate the productivity metric. The controller reads the input values every second, and keeps an exponential moving average of these metrics that spans the last 60 seconds to prevent transient oscillations. The controller uses a simple heuristic: It increases the number of database connections whenever both the productivity and overhead metrics increase, which corresponds to the situation where the CPU has idle time and low DTLB and data cache miss rates. The controller decreases the number of connections whenever the overhead metric increases and the productivity metric stagnates, which corresponds to the situation where the CPU is saturated and the DTLB and data cache miss rates are high.

Figure 2 shows the performance of the baseline system (without the controller) and of the system using reflection (through the use of the controller). The performance of the baseline system increases with the load until it reaches the peak plateau. Then, the performance degrades because of thrashing due to DTLB misses and data cache misses. As for the configuration that uses reflection, the database server is able to sustain peak throughput throughout the overload region by controlling the number of active connections to prevent thrashing.

Although, the controller prevented thrashing in the overload region, the dynamic behavior of the system is still not satisfactory. The controller uses that simple heuristic in an ad-hoc manner rather than a

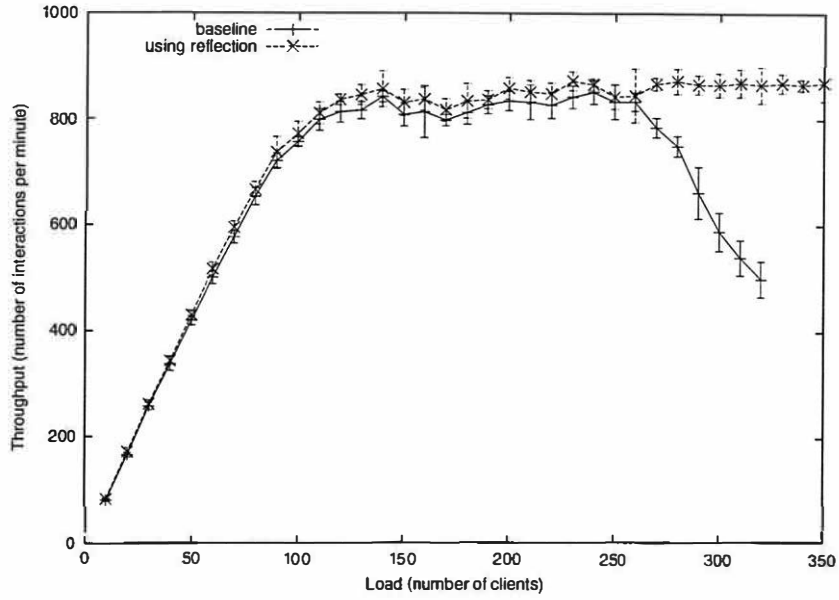


Figure 2: Server throttling for MySQL database server under TPC-W workload.

control-theoretic approach, which would guarantee stability and responsiveness. We believe that it is feaible to apply a control-theoretic approach that prevents thrashing and substantially improves the dynamic behavior of the system as the load changes.

5 Related Work

In this section we briefly mention representative work in areas where similar approaches are used.

Long term schedulers in batch systems have used page fault frequency (PFF) as the objective function for decisions to increase or decrease the multiprogramming level.

Our approach is complementary to Morph [17]. Instead of optimizing the performance by rewriting the binary code, we change the behavior of the system software without rewriting its binary code. We argue that it possible for both approaches to applied simultaneously because Morph-like optimizations optimize the code for a specific hardware or end-user pattern, whereas our approach addresses other performance issues, such as thrashing, which should be handled by specific policies (e.g., limiting the level of concurrency or changing the scheduling policy) rather than binary code optimization.

Douceur and Bolosky [6] used similar techniques to regulate low-importance processes. Our approach strives to maintain maximal performance by adapting the system behavior using both productivity and overhead metrics. This is in contrast to their approach where only productivity (progress) metrics are used to regulate low-importance processes such that they do not affect the execution of other processes.

SEDA [15, 14] presents a staged architecture for Internet servers. Our approach is another point in the design space of building systems that change their behavior adaptively to improve performance. SEDA offers greater control within each stage of a server; however, it requires a complete rewrite of the software. In contrast, our approach gives less control and requires far fewer modifications to the software.

The MAGNET [7] tool tracks OS events and exports information on them to user level. It has been used to identify Linux kernel problems (e.g., Ethernet driver, scheduler anomalies, overheads) and it has been used to analyze and tune applications, including creation of a reflective application.

In the Atlas [16] project, empirical techniques are used to tune the performance of some BLAS and LAPACK routines to provide portable performance.

Bershad et al. [3] used feedback from special hardware to dynamically avoid conflict misses in large direct-mapped caches by reassigning the conflicting virtual memory pages.

In the AppLeS [2] project, an application-level scheduler is used to adaptively and dynamically schedule individual applications on distributed, heterogeneous systems.

6 Summary and Conclusions

We discussed the use of hardware instrumentation mechanisms that are universally available on modern processors and chipsets as a basis for a performance reflection facility. Using this facility, it possible to estimate productivity and overhead metrics. Systems software can use these two metrics to improve its performance. We showed several potential uses: The OS kernel can use the metrics to tune its scheduling decisions. Applications can use these metrics to determine the concurrency level. Finally, we provided a working example for using reflection in server throttling to prevent thrashing.

References

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[2] Fran Berman and Rich Wolski. The AppLeS Project: A Status Report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, May 1997.

[3] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[4] AMD Corporation. AMD Athlon Processor x86 Code Optimization Guide. `www.amd.com`.

[5] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual Symposium on Microarchitecture*, Research Triangle Park, North Carolina, December 1997.

[6] John R. Douceur and William J. Bolosky. Progress-based Regulation of Low-importance Processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[7] Mark K. Gardner, Wu chun Feng, M. Broxton, A Engelhart, and G. Hurwitz. MAGNET: A Tool for Debugging, Analysis and Reflection in Computing Systems. In *Submitted to the third IEEE/ACM International International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

[8] John Mellor-Crummey, Robert Fowler, and Gabriel Marin. HPCView: a tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, New Mexico, October 2001.

[9] Sujay Parekh, Susan Eggers, and Henry Levy. Thread-Sensitive Scheduling for SMT Processors. Technical report, University of Washington, 2002.

[10] Mikael Pettersson. PerfCtr home page. `http://user.it.uu.se/\~{}mikpe/linux/perfctr`.

[11] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[12] The Transaction Processing Council (TPC). TPC-W. `http://www.tpc.org/tpcw`.

[13] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[14] Matt Welsh and David Cluller. Adaptive overload control for busy Internet servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2003.

[15] Matt Welsh, David Cluller, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

[16] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):2–35, January 2001.

[17] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automatic Profiling and Optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

Cassyopia: Compiler Assisted System Optimization

Mohan Rajagopalan Saumya K. Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{*mohan, debray*}*@cs.arizona.edu*

Matti A. Hiltunen Richard D. Schlichting
AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932, USA
{*hiltunen, rick*}*@research.att.com*

Abstract

Execution of a program almost always involves multiple address spaces, possibly across separate machines. Here, an approach to reducing such costs using compiler optimization techniques is presented. This paper elaborates on the overall vision, and as a concrete example, describes how this compiler assisted approach can be applied to the optimization of system call performance on a single host. Preliminary results suggest that this approach has the potential to improve performance significantly depending on the program's system call behavior.

1 Introduction

Execution of a program almost always involves multiple address spaces, whether it executes on a standalone system such as a desktop or a PDA or across multiple machines such as a client-server program or a program based on the Web or the Grid. On a standalone system, user-level address spaces—i.e., processes—request services from the kernel address space using system calls and potentially interact with other user-space processes or system daemons. Programs that span machines are by definition composed of multiple processes, which also interact with kernels, e.g., to exchange messages. Address spaces play a valuable role as protection boundaries, and typically represent units of independent development, compilation, and linking. Largely for these reasons, crossing address spaces—even on the same machine—has considerable execution cost, typically orders of magnitude higher than the cost of a procedure call [11, 12].

Here, we describe an approach to reducing this cost—sometimes dramatically—using compiler optimization techniques. Unlike traditional uses of such techniques that are confined to optimizing within procedures (intra-procedural optimization), across procedures (inter-procedural optimization), or across compilation units (whole-program optimization), our approach focuses on applying these techniques across address spaces on the same or different machines while preserving the desirable features of separate address spaces. The specific focus is on *profiling-based optimization* where the address space crossing behavior of a component (e.g., the system calls made by a process) is profiled and then optimized by reducing the number of crossings or the cost of each crossing. This compiler assisted approach to system optimization is being realized in a system called Cassyopia.

This paper elaborates on this overall vision. We first highlight its application in one context, that of optimizing traditional system call performance on a single host. This work complements existing techniques for system call optimization [5, 6, 11, 14, 15, 16], which focus on optimizing calls in isolation rather than as collections of multiple calls as done here. We then briefly discuss other areas in which these ideas could be applied.

2 Case Study: System Call Clustering

Overview. As an application of the general approach described above, we describe *system call clustering*, a profile-directed approach to optimizing a program's system call behavior. In this approach, execution profiles are used to identify groups of systems calls that can be replaced by a single call implementing their combined functionality, thereby reducing the number of kernel boundary crossings. A key aspect of the approach is that the optimized system calls need not be consecutive statements in the program or even within the same procedure. Rather, we exploit correctness preserving compiler transformations such as code motion, function inlining, and loop unrolling to maximize the number and size of the clusters that can be optimized. The single combined system call is then constructed using a new *multi-call* mechanism that is implemented using kernel extension facilities like those described in [3, 4, 7, 8, 15, 20]. We also introduce an extension to the basic technique called *looped multi-calls*, and illustrate the approach using a simple copy program. Our approach goes beyond earlier work on batching system calls to improve performance [2, 6] by its use of compiler-based techniques to create optimization opportunities and by its orientation towards

optimizing a program's entire system call behavior in a holistic manner.

Clustering mechanisms. We first describe the mechanisms used to realize system call clustering in a traditionally structured operating system. The goal, in addition to reducing boundary crossing costs, is to reduce the number of boundary crossings required. Specifically, we want to extend the kernel to allow the execution of a sequence of system calls in a single boundary crossing. The new mechanism must not compromise protection, transparency or portability, significant advantages provided by the existing system call mechanism. We now look at one such mechanism, the *multi-call* [17].

A multi-call is a mechanism that allows multiple system calls to be performed on a single kernel crossing, thereby reducing the overall execution overhead. Multi-calls can be implemented as a kernel level stub that executes a sequence of system calls. At the application level, the multi-call interface resembles a standard system call and uses the same mechanism to perform the kernel boundary crossing, thereby retaining the desirable features of the system call abstraction. An ordered list of system calls to be executed is passed as a parameter to the multi-call. Each system call in the list is described by its system call number and parameters. Error behavior is preserved by generating the stub so that it returns control to the application level if an error is detected during execution of any of the constituent calls. Also, since the multi-call stub uses the original system call handlers, permissions and parameters are checked as in the original system call.

Modifications to a program to replace a sequence of system calls by a `multicall` are conceptually simple and can be done using a compiler without requiring changes to any other system component (e.g. the linker).

Profiling. Given this mechanism, the issue becomes one of identifying optimization opportunities in the program, both in the sense of identifying sequences of calls that can be replaced by a multi-call and identifying correctness-preserving program transformations that can be used to create such sequences. Profiling does this by characterizing the dynamic system call behavior of a program on a given set of inputs. Operating system kernels often have utilities for generating such traces (e.g., `strace` in Linux), or they can be obtained by instrumenting kernel entry points to write to a log file.

A *system call graph* is then constructed to provide a graphical representation of a collection of such traces for a given program. An example of such a graph for a simple copy program is shown in figure 1.c; the code for this program is in figure 1.a, while 1.b shows the control flow. Each node in this graph represents a system call with a given set of arguments. Consecutive system calls in the trace appear as nodes connected by directed edges indicating the order. The weight of each edge indicates the number of times the sequence appears. This graph forms the basis for compile-time transformations for grouping system calls. The general idea is to find frequently executed sequences of calls in the system call graph; if the corresponding system calls are not syntactically adjacent in the program source, we attempt to restructure the program code so as to make them adjacent, as described below.

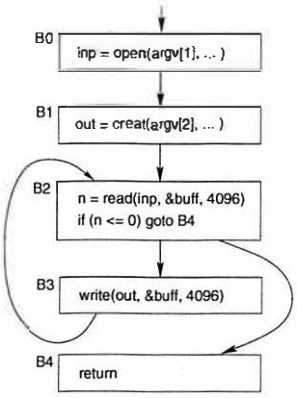
Applying compiler optimizations. The fact that two system calls appear as a sequence in the graph does not, by itself, imply that the system calls can be grouped together. This is because even if two system calls follow each other in the trace, the system calls in the program code may be separated by arbitrary user code that does not include system calls. Replacing these calls by a multi-call would require moving the intervening code into the multi-call, which may compromise safety. Instead, we use compiler techniques like function inlining, code motion, and loop unrolling to transform the program and create sequences of system calls that can be optimized. The use of these standard and well-understood optimization techniques ensures that the transformations are correctness preserving and that the optimized program behaves the same as the original [1]. This also allows tools such as those for checking program safety to be used on the optimized program in the same way as they would for the original. Although code rearrangement is a common compiler transformation, to our knowledge it has not been used to optimize system calls as done here.

A number of program transformations can be used to rearrange the statements in a program to allow system call grouping without affecting the observable behavior of the program. A simple example involves interchanging *independent statements*. Two statements are said to be independent if neither one reads from or writes to any variable that may be written to by the other. Two adjacent statements that are independent and have no externally visible side-effects may be interchanged without affecting a program's observable behavior. This transformation can be used to move two system calls in a program closer to each other, so as to allow them to be grouped into a multi-call. Note that such system calls may actually start out in different procedures, but can be brought together (and hence, optimized) using techniques such as function inlining.

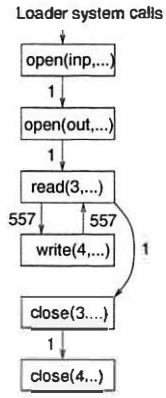
Another useful transformation is *loop unrolling*. In the control flow graph (figure 1.b), the `if` statement in basic block B2 prevents the `read` and the `write` system calls from being grouped together. Programs like FTP,

```
#include <stdio.h>
#include <fcntl.h>

#define N 4096

void main(int argc, char* argv[])
{
  int inp, out, n;
  char buff[N];

  inp = open(argv[1],O_RDONLY);
  out = creat(argv[2],0666);

  while ((n = read(inp,&buff,N)) > 0) {
    write(out,&buff,n);
  }
}
```

(a) Source code



(b) Control flow graph



(c) System call graph

Figure 1: Copy program

encryption programs, and compression programs (e.g., gzip and pzip) exhibit similar control dependencies. In cases like this where the dependency appears within a loop, loop unrolling can sometimes be used to eliminate the dependency. In the case of the copy program in figure 1, for example, unrolling the loop once and combining the footer of one iteration with the header of the next iteration results in the code shown below, with adjacent system calls within the loop that are now candidates for the multi-call optimization:

```
n = read(inp, buff, N);
while (n > 0) {
    write(out, buff, n);
    n = read(inp, buff, N);
}
```

Looped multi-calls. The *looped multi-call* is a variant of the basic multi-call mechanism that repeats the multi-call sequence a specific number of times. It is applicable in the situation where, after other transformations have been applied, the entire body of a loop consists of a single multi-call. In this case, the number of boundary crossings can be reduced to one rather than one per iteration by moving the loop into the kernel. For example, in the copy program, the entire loop can be moved into the kernel using the looped multi-call construct once the write-read sequence is replaced by a multi-call. This optimization is actually a simple extension of traditional *loop invariant motion* [1] applied across address spaces.

Experimental results. A number of experiments have been performed to identify both the potential and actual benefits of this approach. All tests were run on a Pentium II-266 Mhz laptop running Linux 2.4.4-2.

As a baseline, we first measured the cost of a system call versus a procedure call. Table 1 gives the results of these experiments; these results were obtained using the `rdtscl` call, which reads the lower half of the 64 bit hardware counter Read Time Stamp Counter, RDTSC, provided on Intel Pentium processors. These results indicate that clustering even two system calls and replacing them with a multi-call can result in savings of over 300 cycles every time the pair of system calls is executed.

	Entry	Exit
System Call	140 (173-33)	189 (222-33)
Procedure Call	3 (36-33)	4 (37-33)

Table 1: CPU cycles for entry and exit

Table 2 gives the results of applying system call clustering using both the multi-call and the looped multi-call to the copy program shown in figure 1. To do this, the multi-call or looped multi-call was assigned system call number 240 and added as a loadable kernel module. The numbers reported in table 2 were calculated by taking the average of 10 runs on files of 3 sizes ranging from a small 80K file to large files with size around 2MB. The block was chosen as 4096 bytes since it was the page size and hence, the optimal block size for both the optimized and unoptimized versions of the copy program. The maximum benefit in this example is for small and medium file sizes, since the cost of disk and memory operations dominates for larger files.

The second example program is the popular mpeg_play video software decoder [18]. The effects of optimizing this program using our approach are shown in table 3. Although several candidate system call sequences were revealed by profiling, only one was optimized since the

File Size	Original	Multi-call		Looped Multi-Call	
	Cycles (10^6)	Cycles (10^6)	% Savings	Cycles (10^6)	% Savings
80K	0.3400	0.3264	4%	0.3185	6.3%
925K	4.371	4.235	3.1%	4.028	7.8%
2.28M	10.93	10.65	2.6%	10.37	5.2%

Table 2: Optimization of a copy program with block size of 4096

others existed partially or completely in the X-windows libraries used by the player. The program was executed using different input files taken from [13] with sizes varying from 4.7MB to 15MB. Overall, our approach shows a more dramatic effect than for the copy program, largely because the system calls here are not I/O bound as was the case for copy. In addition to the savings in CPU cycles, this optimization also improved the frame-rate and performance of `mpeg_play`. Specifically, there was an average 25% improvement in the frame rate and 20% reduction in execution time across all file sizes.

More details on these and other examples can be found in [17].

3 Other Compiler Assisted Techniques

The multi-call mechanism can be extended further to include code other than the system calls, error checking, and loops in the multi-call. Specifically, we can extend the basic code-motion transformations to identify a *clusterable region*, possibly containing arbitrary code, that can then be added to the body of a multi-call. Optimization techniques like dead-code elimination, loop invariant elimination, redundancy elimination, and constant propagation can then be applied to optimize the program. For example, the data transformation code in programs such as compression or multimedia encoding/decoding can be included in the multi-call.

Another avenue of optimization is to replace general purpose code in the kernel by compiler-generated case-specific code in user-space. Examples of such general code are the register saves and restores executed by the kernel before and after each system call. Since the kernel does not know which registers are actually used by the application process, it must save and restore all of them. This can be quite expensive on processors with a large number of registers. However, the compiler has this information, and it can therefore generate specialized user-space code for saving and restoring registers. Simulations using this strategy for a 3 parameter `read` system call on the Intel StrongARM processor show up to 20% reduction in the number of cycles required to enter the kernel. Other such examples include the general permission checking performed by each system call. Note that both of the above extensions require use of a trusted compiler for safety reasons.

The profiling and compiler-based optimization can also be used to enable controlled information sharing between address spaces. Traditionally, components in different address spaces optimize their internal behavior not knowing what type of interactions will be received from other address spaces. For example, the operating system conserves battery power by switching hardware devices such as the CPU, display, hard disk, and wireless cards into power-saving modes based on a period of inactivity. These policies are generally based on statistical models of application behavior that attempt to predict future (in)activity based on patterns of past activity. Because of their stochastic nature, they can be quite inaccurate for individual applications, and result in significant performance overheads [21, 22, 23]. However, by carefully exposing some of the components internal state to other address spaces using a *translucent boundary API*, each address space can optimize its behavior to better match the requirements of other system components, and hence aim for a global optimum. The profiling and compiler techniques can be used to collect and generate the information at the application's translucent boundary API to the kernel. The same approach can be used for compiler assisted scheduling, where an adaptive scheduler can fine-tune the scheduling policy based on the processes running in the system and their requirements. The compiler could place "yield" points within the body of the program to indicate schedulable regions and changes in requirements. Conversely, if the kernel exposes changes in the state of an existing resource, e.g., a reduction in CPU speed to conserve power, the application process may be able to adapt its internal algorithms [9] to degrade the service gracefully while still satisfying user requirements.

Finally, note that the same principles can be applied to any address space crossing, including distributed programs where the "boundary crossing" cost involving network communication may have a delay of tens or hundreds of milliseconds. In particular, clustering multiple remote procedure calls (or remote method invocations

Size	CPU Cycles (10^9)		%
	Original	Optimized	Savings
4.7M	23.75	21.74	8.47%
9.5M	63.65	52.09	18.17%
9.5M	31.00	21.70	30.00%
10.3M	51.51	41.12	20.17%
15.1M	60.18	52.10	13.42%

Table 3: Optimization of mpeg_play using multi-calls

in distributed object systems such as CORBA and Java RMI) can lead to significant savings [25]. Furthermore, more general code movement techniques such as moving client code to the server or server code to the client when appropriate can also be used [24]. Note that the object migration techniques used in systems such as Emerald [10] have the same goal, but without the systematic support provided by our profiling and compiler techniques.

4 Concluding Remarks

Our current work is aimed at integrating these optimizations into the PLTO binary rewriting tool [19], and then using the tool to test further the effectiveness of the approach. A number of potential targets have been identified ranging from utility programs like gzip and pzip, to web servers and database applications. We also intend to explore the applicability of these techniques for small mobile devices. Independent of the savings in CPU cycles, we believe our approach will yield significant energy savings in this context, greater in fact than what the reduction in CPU cycles would imply.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] F. Ballesteros, R. Jimenez, M. Patino, F. Kon, S. Arevalo, and R. Campbell. Using interpreted CompositeCalls to improve operating system services. *Software - Practice and Experience*. 30(6):589–615, May 2000.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer. M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort. CO, Dec 1995.

[4] R. Campbell and S. Tan. μ-Choices: An object-oriented multimedia operating system. In *Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA*, May 1995.

[5] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, Dec 1993.

[6] A. Edwards, G. Watson, J. Lumley, D. Banks, and C. Dalton. User space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *SIGCOMM*, Aug 1994.

[7] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference, Monterey, CA, USA*, June 1999.

[8] B Henderson. Linux loadable kernel module. HOWTO. http://www.tldp.org/HOWTO/Module-HOWTO/, Aug 2001.

[9] M. Hiltunen and R. Schlichting. A model for adaptive fault-tolerant systems. In K. Echtle, D. Hammer, and D. Powell. editors, *Proceedings of the 1st European Dependable Computing Conference (Lecture Notes in Computer Science 852)*, pages 3–20, Berlin, Germany, Oct 1994.

[10] E. Jul, H. Levy, N. Hutchinson. and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb 1988.

[11] J. Mauro and R. McDougall. *Solaris Internals-Core Kernel Architecture*, pages Section 2.4.2 (Fast Trap System Calls), 46–47. Sun Microsystems Press, Prentice Hall, 2001.

[12] J. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, 1991.

[13] Technical University of Munich. http://www5.in.tum.de/-forschung/visualisierung/duenne_gitter.html.

[14] C. Poellabauer, K. Schwan, and R. West. Lightweight kernel/user communication for real-time and multimedia applications. In *NOSSDAV'01*, Jun 2001.

[15] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM*

Symposium on Operating Systems Principles (SOSP'95), pages 314–324, Copper Mountain, CO, Dec 1995.

[16] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[17] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. System call clustering: An automated approach to system call optimization. Technical report, The University of Arizona, March 2003.

[18] L. Rowe, K. Patel, B. Smith, S. Smoot, and E. Hung. Mpeg video software decoder, 1996. http://bmrc.berkeley.edu/mpeg/mpegplay.html.

[19] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO : A link time optimizer for the Intel IA32 architecture. In *Proceedings of Workshop on Binary Rewriting*, Sept 2001.

[20] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Operating Systems Design and Implementation*, pages 213–227, 1996.

[21] T. Simunic, L. Benini, and G. DeMicheli. Event-driven power management of portable systems. *IEEE Transactions on Computer Aided Design*, July 2001.

[22] T. Simunic, L. Benini, P. Glynn, and G. DeMicheli. Dynamic power management of laptop hard disk. *DATE*, 2000.

[23] T. Simunic, H. Vikalo, P. Glynn, and G. DeMicheli. Energy efficient design of portable wireless systems. *ISPLED*, 2000.

[24] D. Waugaman and R. Schlichting. Using code shipping to optimize remote procedure call. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 17–24, Las Vegas, NV, Jul 1998.

[25] Q.Y. Zondervan. Increasing cross-domain call batching using promises and batched control structures. Technical Report MIT/LCS/TR-658, 1995.

Cosy: Develop in User-Land, Run in Kernel-Mode

Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok
Stony Brook University

Abstract

User applications that move a lot of data across the user-kernel boundary suffer from a serious performance penalty. We provide a framework, Compound System Calls (Cosy), to enhance the performance of such user-level applications. Cosy provides a user-friendly mechanism to execute the data-intensive code segment of the application in the kernel. This is achieved by aggregating the data-intensive system calls and the intermediate code into a compound. This compound is executed in the kernel, avoiding redundant data copies.

A Cosy version of GCC makes the formation of a Cosy compound simple. Cosy-GCC automatically converts user-defined code segments into compounds. To ensure the security of the kernel, we use a combination of static and dynamic checks. We limit the execution time of the application in the kernel by using a modified preemptible kernel. Kernel data integrity is assured by performing necessary dynamic checks. Static checks are enforced by Cosy-GCC. To study the performance benefits of our Cosy prototype, we instrumented applications such as `grep` and `ls`. These application showed an improvement of 20–80%. Our current work focuses on faster and secure execution of entire programs in the kernel without source code modification.

1 Introduction

User applications like HTTP, FTP, and mail servers involve copying a significant amount of data across the user-kernel boundary. It is well understood that this cross-boundary data movement is expensive. It can reduce the overall performance of the application by two orders of magnitude [11]. For applications like `ls` that invoke a large number of small system calls, performance is hampered by the time wasted in context switching.

A solution to improve performance of such applications is to move the bottleneck code segment, the segment involving cross-boundary data movement, into the kernel [1, 5]. Applications written for VINO [5] and SPIN [1] have supported this idea. Exokernels [3] take a different approach by allowing user processes to describe the on-disk data structures and methods to implement them. The common problem with these approaches is that they do not fit in the framework of current commodity operating systems. Others attempted to improve the performance but at the cost of safety [8]. Software fault isolation provides a secure mechanism to execute untrusted code, but performance benefits are often lost [9]. Other solutions to improve performance include `sendfile`, NFSv4 [6], and smart storage [7]. Sendfile provides a way to avoid data copies for network-specific applications. NFSv3 adds some new calls like `READDIRPLUS` to reduce the overhead due to a `readdir` followed by several `stat` calls. Smart storage is limited to improving disk I/O and can not be extended to networks. These approaches are successful but they are not extensible.

We present Cosy as a generic solution to safely execute bottleneck code segments of user applications in the kernel. Cosy exploits zero-copy techniques and code aggregation to achieve better performance without reducing the security. Cosy extracts the system calls and the intermediate code from the bottleneck code segment and encodes them to create a *compound*. A *compound* is formed by aggregation of system calls, programmatic constructs (e.g., `if-else` and `while`), and user specified functions. This compound is then passed to the kernel via a new system call (`cosy_run`) which decodes it and executes the decoded elements in the compound intelligently to avoid redundant data copies.

To facilitate automatic generation of compounds we provide Cosy-GCC, a modified version of GCC, which makes writing Cosy compounds simple. The user just needs to mark the bottleneck region and Cosy-GCC converts it into a compound at compile time. Cosy-GCC also finds data dependencies among Cosy statements and encodes this information into the compound. This information is used by the kernel while executing the compound to avoid data copies.

Systems that allow arbitrary user code to execute in kernel mode must address security and protection issues: how to avoid buggy or malicious code from corrupting data, accessing protected data, or crashing the kernel. Securing such code often requires costly runtime checking [8]. Cosy uses a combination of static and dynamic approaches to assure safety in the kernel. Cosy explores various hardware features along with software techniques to achieve maximum safety without adding much overhead.

The rest of this paper is organized as follows. Section 2 describes the design of our system. Section 3 presents an evaluation of our Cosy prototype. We conclude in Section 4.

2 Design

The main motivation behind Cosy is to achieve maximum performance with minimal user intervention, and without compromising security. The primary design goals were as follows:

Performance We exploit several zero-copy techniques at various stages to enhance the performance. We make use of shared buffers between user and kernel space for fast cross-boundary data exchange.

Safety We make use of various security features including kernel preemption and x86 segmentation to assure a robust safety mechanism. We make use of a combination of static and dynamic checks to assure safety in the kernel without adding run-time overheads.

Simplicity We have automated the formation and execution of the compound so that it is almost transparent to the end user. Thus it is simple to write new code as well as modify existing code to use Cosy. The Cosy framework is extensible and adding new features to it is not difficult.

2.1 Architecture

Cosy executes compounds of system calls in the kernel. Often only small sections of code suffer from cross-boundary communication. The first step while using Cosy is to identify these *bottleneck* code segments. A bottleneck code segment is transformed into a compound by identifying and aggregating the system calls, arithmetic, assignment operations, loops, and function calls. To facilitate the formation and execution of a compound, Cosy provides three components: the Cosy kernel extension, Cosy-GCC, and Cosy-Lib. These components communicate using two shared buffers. The *compound buffer* is a shared buffer used to encode the compound. The *fast buffer* is another buffer used to facilitate zero-copy within system calls that share arguments. We look at the individual components and the internals of the compound buffer in the following subsections.

2.1.1 Kernel Extension for Cosy

The Cosy kernel extension exposes three system calls to the user space components.

- **cosy_init** allocates the two shared buffers that are used by the user and kernel to exchange the encoded compound and the results.
- **cosy_run** decodes the compound from the compound buffer and executes the decoded instructions one by one.
- **cosy_uninit** frees any Cosy resources for the current process including the shared buffers.

Each Cosy-enabled process has its own shared buffers.

2.1.2 Cosy-GCC

Currently, the application programmer just needs to identify the bottleneck code segment in the application and mark it in the standard C program using the markers provided by Cosy (COSY_START and COSY_END). Usually, no modifications are needed to the code within these markers. The only constraint is that all the instructions within the marked block should be supported under Cosy-GCC. Cosy-GCC parses the code and if all the instructions within the segment are supported then it modifies the marked code. It also inserts a `cosy_run` at the end of the marked segment. The code is modified in such a way that during execution, the modified code forms a compound and the `cosy_run` at the end informs the Cosy kernel extension to execute this recently-formed compound. Cosy-GCC also maintains a symbol table of labels for Cosy calls. This symbol table is used to find out any interdependency among the arguments of compounded calls. The information about interdependencies is also encoded in the compound buffer. The Cosy kernel extension uses this information to avoid data copies. The symbol table is also used to resolve the jump labels.

Cosy supports loops (e.g., `for`, `do-while`, and `while`), conditional statements (e.g., `if`, `switch`, and `goto`), simple arithmetic operations (e.g., increment, decrement, assignment, add, and subtract) and system calls within a marked code segment. Cosy also provides an interface to execute a piece of user code in the kernel. Applications like `grep`, volume rendering [10], and checksumming are the main motivation behind adding this support. These applications read large amounts of data in chunks and then perform a unique operation on every chunk. To benefit such applications, Cosy provides a secure mechanism to call a user supplied function from within the kernel.

In order to assure the secure execution of the code in the kernel, we restrict Cosy-GCC to support a subset of the C-language. Cosy-GCC ensures there are no unsupported instructions within the marked block, so complex code may need some small modifications to fit within the Cosy framework. This subset is carefully chosen to support different types of code in the marked block, thus making Cosy useful for a wide range of applications.

2.1.3 Cosy-Lib

Cosy-Lib provides a set of utility functions to insert entries into a compound. Cosy-GCC, while compiling a marked segment, inserts calls to these utility functions. So generally the functioning of Cosy-Lib is entirely transparent to the user as it is done by Cosy-GCC. But it is also possible for programmers to manually create a compound using these utility functions. It is possible to design complex or hand-optimized compounds using this facility.

2.1.4 Compound Buffer

In this section we discuss the internal representation of Cosy compound (see Figure 1). A compound is a set of entries belonging to one of the following types:

- System calls
- Arithmetic operations
- Variable assignments
- `while`, `do-while`, and `for` loops
- User provided functions
- Conditional statements
- `switch` statements
- Labels
- `gotos` (unconditional branches)

The first component of the compound buffer is the global header. It contains the number of entries in the compound, the length of the compound in words, and the

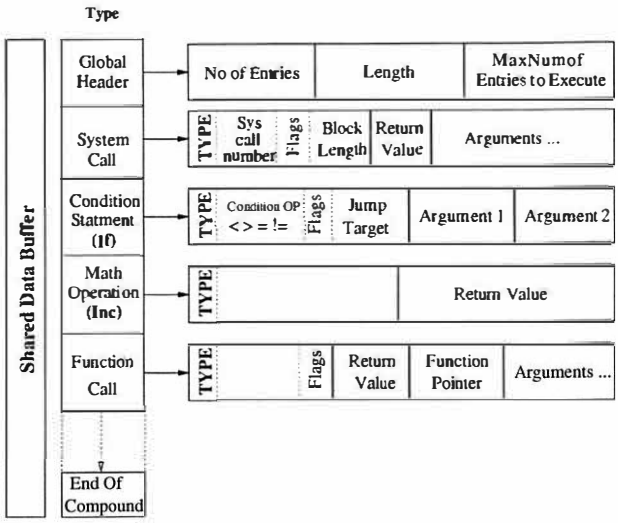


Figure 1: An Example of the Structure of a Cosy Compound

maximum number of entries to execute. This entry ensures protection against unending compounds. The rest of the compound contains a set of entries with a local header followed by a number of arguments.

Each type of entry has a different structure for the local headers. Each local header has a `type` field, which uniquely identifies the entry type. Depending on the type of the entry, the rest of the arguments are analyzed. For example, if the entry is of the type "system call" then the local header contains the system call number and flags. The flags indicate whether the argument is the actual value or it is a reference to the output of another entry. The latter occurs when there are argument dependencies. If it is a reference, then the actual value is retrieved from the reference address. The local header is followed by a number of arguments necessary to execute the entry. If the execution of the entry returns any value (as in system calls, math operations, and function calls) then one position is reserved to store the result of the execution. Conditional statements affect the flow of the execution. The header of a conditional statement specifies the operator and the next instruction executed, if the condition is satisfied. Cosy-GCC resolves dependencies among the arguments and the return values, the correspondence between the label and the compound entry, and forward references for jump labels.

The overall performance of the framework depends on the efficiency of the decoder. Hence we have used several techniques like lazy caching and fast system call invocation to optimize the decoder. The first time any entry is decoded, we store the decoded value in a hash table. This makes it possible to decode an entry only once; subsequent accesses use the hashed entry. Another optimization is achieved by pushing the arguments of the system call directly onto the stack since they are packed in the Compound in the same order as they should appear on the Kernel's Stack. This makes system call invocation faster. A small piece of assembly code helps to achieve this faster invocation of system calls.

2.2 Shared Data Buffer

In this section we explain various ways that Cosy exploits zero-copy techniques by the aggregation of data-intensive code. Much work has already been done on zero-copy techniques. Cosy uses similar techniques but it tries to combine multiple techniques and provide a uniform interface to the user. Depending on the type of application, different zero-copy techniques are employed.

Cosy modifies the behavior of `copy_from_user` (copies data from the user address space into the kernel address space) and `copy_to_user` (the converse of `copy_to_user`) to enable zero-copy when the user is not interested in getting the data back into the user space. For example, when there is data dependency between a read and a following write call, Cosy uses the fast buffer to avoid the redundant copy.

Cosy also supports special versions of system calls that are commonly used. The extensive use of these system calls justifies the creation of zero-copy equivalents. We currently support zero-copy versions of `cosy_read`, `cosy_write`, and `cosy_stat`. Cosy-GCC uses these system calls automatically.

2.3 Safe Execution of a Compound

Cosy makes use of a combination of static and dynamic checks to ensure safe execution of compound. Cosy is not vulnerable to bad arguments when executing the system calls on behalf of a user process. The system call invocation by the Cosy kernel module is the same as a normal process and hence all the necessary checks are observed. However, when executing a user-supplied function, more safety precautions are needed. We describe two interesting Cosy safety features in the next sections: a preemptive kernel to avoid infinite loops, and x86 segmentation to protect kernel memory.

2.3.1 Kernel Preemption

One of the critical problems that needs to be handled while executing a user function in the kernel is to limit its execution time. To handle such situations, Cosy uses a preemptible kernel. A preemptible kernel allows scheduling of processes even when they are running in the context of the kernel. So even if a Cosy process causes an infinite loop, it is eventually scheduled out just like a normal user process. Every time a Cosy process is scheduled out, Cosy interrupts and checks the running time of the process inside the kernel. If this time has exceeded the maximum allowed kernel time, then the process is terminated. We modified the scheduler behavior to add this check for Cosy processes. The added code is minimal and is executed only for Cosy processes and hence does not affect the overall system performance. The limit on the amount of kernel time is kept sufficiently high. This high limit is not a security concern. The process even if running in kernel mode could be scheduled out and hence it does not starve other processes.

2.3.2 x86 Segmentation

To assure the secure execution of user-supplied functions in the kernel, we use the Intel x86 segmentation feature. We support two approaches.

The first approach is to put the entire user function in an isolated segment but at the same privilege level. The static and dynamic needs of such a function are satisfied using memory belonging to the same isolated segment. This approach assures maximum security, as any reference outside the isolated segment generates a protection fault. Also, if we use two non-overlapping segments for function code and function data, concerns due to self-modifying code vanish automatically. However, to invoke a function in a different segment involves overhead. Before making the function call, the Cosy kernel extension saves the current state so it is able to resume execution later. Saving the current state and restoring it back is achieved by using the standard task-switching macros, `SAVE_ALL` and `RESTORE_ALL`, with some modifications. These macros involve around 12 assembly `pushl` and `popl` instructions, each. So if the function is small and it is executed a large number of times, this approach could be costly due to the added overhead of these two macros. The important assumption here is that even if the code is executing in a different segment, it still executes at the same privilege level as the kernel. Hence, it is possible to access resources exposed to this isolated segment, without any extra overhead. Currently, we allow the isolated code to read only the shared buffer, so that the isolated code can work on this data without any explicit data copies.

The second approach uses a combination of static and dynamic methods to assure security. In this approach we restrict our checks to only those that protect against malicious memory references. This is achieved by isolating the function data from the function code by placing the function data in its own segment, while leaving the function code in the same segment as the kernel. In Linux, all the data references are resolved using the `ds` segment register, unless a different segment register is explicitly specified. In this approach, all accesses to function data are forced to use a different segment register than `ds` (`gs` or `fs`). The segment register (`gs` or `fs`) points to the isolated data segment, thus allowing access only to that segment; the remaining portion of the memory is protected from malicious access. This is enforced by having Cosy-GCC append a `%gs` (or `%fs`) prefix to all memory references within the function. This approach involves no additional runtime overhead while calling such a function, making it very efficient. However, this approach has two limitations. It provides little protection against self modifying code, and it is also vulnerable to hand-crafted user functions that are not compiled using Cosy-GCC. We are exploring compiler techniques such as self certifying code [5] to address the above concerns.

2.4 Cosy Example

In this section we illustrate some simple examples of code using Cosy. We write them once using Cosy-GCC and then without using Cosy-GCC. To improve clarity and conserve space we do not include any error checking. In these examples, we assume that the system calls without any error checking always succeed.

The following code is a simple file copy program. It reads from input file `ifile` and copies the data read to generate a duplicate file `ofile`:

```
1  cosy_init();
2
3  COSY_START();
4  ifd = open(ifile, O_RDONLY);
5  ofd = open(ofile, O_WRONLY);
6
7  do {
8     rlen = read(ifd, buf, 4096);
9     wlen = write(ofd, buf, rlen);
10 } while(wlen == 4096);
11 COSY_END();
12 cosy_uninit();
```

In the above program we can see that the code is almost unchanged except four new instructions. When the above code is compiled using Cosy-GCC it takes the following form:

```
1  cosy_init();
2
3  cosy_start();
4  cosy_open(&ifd, ifile, O_RDONLY);
5  cosy_open(&ofd, ofile, O_WRONLY);
6
7  cosy_do();
8     cosy_read( &rlen, ifd, buf, 4096);
9     cosy_write(&wlen, ofd, buf, 4096);
10
11 cosy_while(wlen, "==", 4096);
12 cosy_run();
13 cosy_uninit();
```

Line 1 allocates the shared buffers for the process. Line 3 clears the compound buffer. Lines 4--11 add entries into the compound. It includes a while loop around `read` and `write`. Line 12 instructs the Cosy kernel module to execute this compound. Finally, line 13 releases any buffers that are owned by this process.

3 Current Status

In this section we present benchmarks of our prototype that show the effectiveness of Cosy in improving performance of various applications. We used the following three configurations to compare the results:

1. **VAN:** This is a generic setup. This configuration does not use Cosy.
2. **COSY:** This is the Cosy configuration. It uses the Cosy framework to form and execute a compound, but does not use zero-copy techniques.
3. **COSY-FAST:** This configuration makes use of compounds and the zero-copy system calls provided by Cosy.

We performed our tests on a Intel Pentium-IV 1.7GHz machine with 64MB of RAM and a 7200 RPM 20GB ATA/100 hard drive. We repeated each test 20 times and the observed standard deviations were less that 5%. To evaluate the performance of Cosy, we report results of three benchmarks: a database simulation, `ls`, and `grep`.

Database Simulation In this benchmark we evaluate the benefits of Cosy for a database-like application. We wrote a program that seeks to random locations in a file and then reads and writes to it. The total number of reads and writes is six million. The ratio of reads to writes we chose is 2:1, matching pgmeter's [2] database workload.

We used all three configurations VAN, COSY, and COSY_FAST. We ran the benchmark for increasing file sizes. We also ran this benchmark with multiple processes to determine the scalability of Cosy in a multiprocess environment. In call test, we kept the number of transactions constant at six million.

Both versions of Cosy perform better than VAN. COSY_FAST shows a 64% improvement, while COSY shows a 26% improvement in the elapsed time as seen in Figure 2. COSY_FAST is better than COSY by 38%. This additional benefit is the result of the zero-copy savings. The improvements achieved are stable even when the working data set size exceeds system memory bounds, since the I/O is interspersed with function calls.

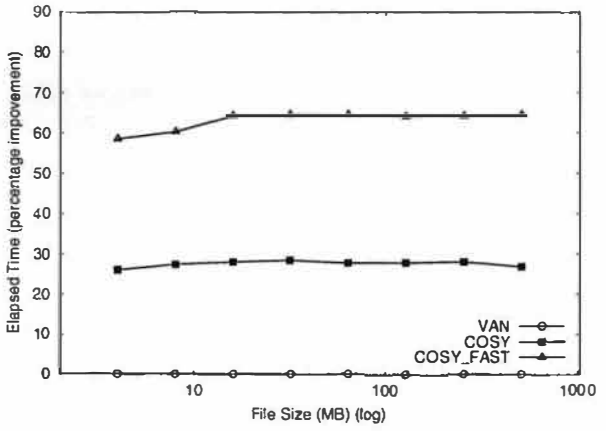


Figure 2: Elapsed and system time percentage improvements for the Cosy database benchmark (over VAN).

We also tested the scalability of Cosy, when multiple processes are modifying a file concurrently. We repeated the database test for 2 and 4 processes. We kept the total number of transactions performed by all processes together fixed at six million. We compared these results with the results observed for a single process. We found the results were indistinguishable and they showed the same performance benefits of 60–70%. This demonstrates that Cosy is beneficial in a multi-threaded environment as well.

Is We instrumented a Cosy `ls -l` program and compared it with the generic `ls -l` program. We use three configurations VAN, COSY and COSY-FAST for this benchmark.

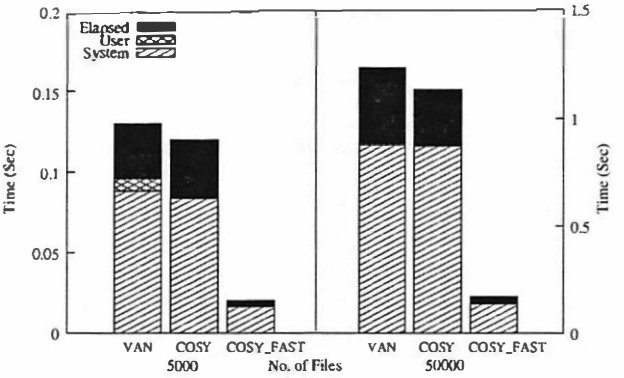


Figure 3: Elapsed, system, and user times for the Cosy `ls -l` benchmark. Note that the left and right sides of the graph use different scales.

For each configuration we ran this benchmark for 5000 and 50000 files and recorded the elapsed, system, and user times. We unmounted and remounted the file system between each test to ensure cold cache.

Figure 3 shows the system, user, and elapsed times taken by VAN, COSY, and COSY_FAST for listing of 5000 and 50000 files. COSY shows an 8% improvement over VAN. COSY_FAST performs 85% better than VAN for both the cases. The results indicate that Cosy performs well for small as well as large workloads, demonstrating its scalability.

grep To find out the effect of Cosy on data-intensive applications we instrumented `grep` with Cosy. This benchmark reads the files in a specified directory and executes a user provided function that searches the buffer for a given string. We recorded results for increasing sizes of data. We used all the three configurations mentioned at the start of this section.

The Cosy version of `grep` with zero-copy performs 20% better than the normal version (see Figure 4). The nonzero-copy version also shows improvement of 15%. The 5% difference between the two flavors of Cosy justify the use of special zero-copy calls to further improve performance. This improvement substantiates our claim that moving the user function into the kernel can give us additional performance benefits.

4 Conclusions

In our work we introduce a safe yet efficient mechanism to execute bottleneck code segments of user applications, in

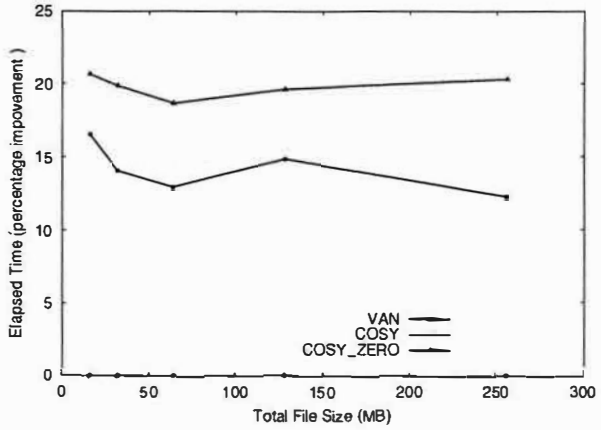


Figure 4: Elapsed time % improvement for `grep`

kernel mode. We provide various zero-copy techniques that benefit different user level applications. Thus we show the applicability of Cosy under different environments. For user convenience we provide an automated mechanism to form a compound out of user-marked code. The marked code can contain loops, system calls, arithmetic operations and even some simple functions. Thus a wide range of code can be moved to the kernel transparently. Cosy supports a subset of a widely-used language, namely C, making Cosy easy to work with. We have prototyped Cosy on Linux, which is a commonly-used operating system. Many widely used user applications exist for Linux. We show performance improvements for these commonly-used applications without compromising safety.

Our benchmarks prove the usefulness and effectiveness of compound system calls. We show a speed improvement of 20-80% depending on the type of application.

4.1 Future Work

The Cosy work is an important step toward the ultimate goal of being able to execute unmodified Unix/C programs in kernel mode. The major hurdles in achieving this goal are safety concerns.

We plan to explore heuristic approaches to authenticate untrusted code. The behavior of untrusted code will be observed for some specific period and once the untrusted code is considered safe, the security checks will be dynamically turned off. This will allow us to address the current safety limitations involving self-modifying and hand-crafted user-supplied functions.

Intel's next generation processors are designed to support security technology that will have a protected space in main memory for a secure execution mode [4]. We plan to explore such hardware features to achieve secure execution of code in the kernel with minimal overhead.

To extend the performance gains achieved by Cosy, we are designing an I/O-aware version of Cosy. We are exploring various smart-disk technologies [7] and typical disk access patterns to make Cosy I/O conscious.

5 Acknowledgments

This material is based upon work supported by the National Science Foundation CAREER award Grant No. 0133589.

The work described in this paper is Open Source Software and is available for download from `ftp://ftp.fsl.cs.sunysb.edu/pub/cosy`.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[2] R. Bryant, D. Raddatz, and R. Sunshine. PenguinoMeter: A New File-I/O Benchmark for Linux. In *Proceedings of the 5th Annual Linux Showcase & Conference*, pages 5–10, Oakland, CA, November 2001.

[3] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. Technical Report CMU-CS-00-117, Carnegie Mellon University, March 2000.

[4] H. B. Pedersen. Pentium 4 successor expected in 2004. *Pcworld*, October 2002. www.pcworld.com/news/article/0,aid,105882,00.asp.

[5] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the VINO kernel. Technical Report TR-34-94, EECS Department, Harvard University, 1994.

[6] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3010, Network Working Group, December 2000.

[7] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of First USENIX conference on File and Storage Technologies*, March 2003.

[8] T. Maeda. Safe Execution of User programs in kernel using Typed Assmebly language. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml, 2002.

[9] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 203–216, Asheville, NC, December 1993. ACM SIGOPS.

[10] C. Yang and T. Chiueh. I/O conscious Volume Rendering. In *IEEE TCVG Symposium on Visualization*, May 2001.

[11] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.

Why can't I find my files? New methods for automating attribute assignment

Craig A. N. Soules, Gregory R. Ganger
Carnegie Mellon University

Abstract

Attribute-based naming enables powerful search and organization tools for ever-increasing user data sets. However, such tools are only useful in combination with accurate attribute assignment. Existing systems rely on user input and content analysis, but they have enjoyed minimal success. This paper discusses new approaches to automatically assigning attributes to files, including several forms of context analysis, which has been highly successful in the Google web search engine. With extensions like application hints (e.g., web links for downloaded files) and inter-file relationships, it should be possible to infer useful attributes for many files, making attribute-based search tools more effective.

1 Introduction

As storage capacity increases, the amount of data belonging to an individual user increases accordingly. Soon, storage capacity will reach a point where there will be no reason for a user to ever delete old content – in fact, the time required to do so would be wasted. The challenge has shifted from deciding what to keep to finding particular information when it is desired. To meet this challenge, we need better approaches to personal data organization.

Today, most systems provide a tree-like directory hierarchy to organize files. Although this is easy for most users to reason about, it does not provide the flexibility required to scale to large numbers of files. In particular, a strict hierarchy provides only a single categorization with no cross-referencing information.

To deal with these limitations, several groups have proposed alternatives to the standard directory hierarchy [5, 9, 11]. These systems generally assign attributes to files, providing the ability to cluster and search for files by their attributes. An attribute can be any metadata that describes the file, although most systems use keywords or ⟨category, value⟩ pairs. The key challenge is assigning useful, meaningful attributes to files.

To assign attributes, these systems have suggested two largely unsuccessful methods: user input and content analysis. Although users often have a good understanding of the files they create, it can be time-consuming and unpleasant to distill that information into the right set of keywords. As a result, users are understandably reluctant to do so. On the other hand, content analysis takes none of the user's time, and it can be performed entirely in the background to eliminate any potential performance penalty. Unfortunately, the complexity of language parsing, combined with the large number of proprietary file formats and non-textual data types, restrict the effectiveness of content analysis.

A complementary alternative to these methods is context analysis. Context analysis gathers information about the user's system state while creating and accessing files, and uses it to assign attributes to those files. This can be useful in two ways. First, such context is often related to the content of a file. For example, a user may read an email about a friend's dog and then look at a picture of that same dog. Second, the context may be what a user remembers best when searching for some files. For example, the user may remember what they were working on when they downloaded a file, but not what they named the file.

This paper discusses two categories of context analysis: access-based context analysis and inter-file context analysis. The first gathers information about the state of the system when a user accesses a file. The second propagates attributes among related files. Combining these methods with existing content analysis and user input will increase the information available for attribute assignment.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. Section 3 describes access-based context analysis. Section 4 discusses recognition and use of inter-file relationships. Section 5 presents some initial findings. Section 6 discusses some challenges facing this work, and ideas on how to approach them.

2 Background

Users already have difficulty locating their files. There exist a variety of tools for locating files by searching

through directory hierarchies, but they don't solve the problem. Several groups have proposed attribute-based naming systems that rely on user input and content analysis to gather attributes, but they remain largely unused. Web search engines, however, have found greater success obtaining attributes by combining content analysis with context analysis. This section discusses common approaches to file organization, proposed systems, and relevant web search-engine approaches.

2.1 Directory Hierarchies

There are three key factors that limit the scalability of existing directory hierarchies. First, files within the hierarchy only have a single categorization. As the categories grow finer, choosing a single category for each file becomes more and more difficult. Although linking (giving multiple names to a file) provides a mechanism to mitigate this problem, there exists no convenient way to locate and update a file's links to reflect re-categorization (since they are unidirectional). Second, much information describing a file is lost without a well-defined and detailed naming scheme. For example, the name of a family picture would likely not contain the names of every family member. Third, unless related files are placed within a common sub-tree, their relationship is lost.

One way to try and overcome these limitations is to provide tools to search through these hierarchies. Today, on UNIX systems, many users locate files via tools such as *find* and *grep*. These tools provide the ability to search throughout a hierarchy for given text within a file, providing rudimentary content analysis. *Glimpse* [14] is a system that provides similar functionality, but utilizes an index to improve the performance of queries. Microsoft Windows' search utility provides a similar indexing service using filters to gather text from well-known file formats (e.g., Word documents). Going a step further, systems such as *LXR* and *CScope* [22], perform content analysis on well-known file formats to provide some attribute-based searching features within a hierarchy (e.g., locating function definitions within source code).

2.2 Proposed Systems

To go beyond the limitations of directory hierarchies, several groups have proposed extending file systems to provide attribute-based indexing. For example, BeFS extends the directory hierarchy by adding a new organizational structure for indexing files by attribute [8]. The system takes a set of ⟨file, keyword⟩ pairings and creates an index allowing fast lookup of an attribute value to return the associated file. This structure is useful for files that have a set of well-known attributes on which to index (e.g., ⟨email message, sender⟩).

The semantic file system [9] provides a way to assign generic ⟨category, value⟩ pairings to files, increasing the scope of their namespace. These attributes are assigned either by user input or by file content analysis. Content analysis is done by a set of *transducers* that each understand a single well-known file format. Once attributes are assigned, the user can create virtual directories that contain links to all files with a particular attribute. The search can be narrowed by creating further virtual subdirectories.

Several groups have explored other ways of merging hierarchical and attribute-based naming schemes. Sechrest and McClennen [21] detail a set of rules for constructing various mergings of hierarchical and flat namespaces using Venn diagrams. Gopal [10] defines five goals for merging hierarchical name spaces with attribute-based naming and evaluates a system that meets those goals.

Other groups have looked at the problem of providing an attribute-based naming scheme across a network of computers. Harvest [3] and the Scatter/Gather system [5] provide a way to gather and merge attributes from a number of different sites. The Semantic Web [1] proposes a framework for annotating web documents with XML tags, providing applications with attribute information that is currently not available.

These systems provide a number of interesting variations on attribute-based naming. But they all rely upon user input and content analysis to provide useful attributes, with limited success.

2.3 Context Analysis

Early web search-engines, such as Lycos [15], relied upon user input (user submitted web pages) and content analysis (word counts, word proximity, etc.). Although valuable, the success of these systems has been eclipsed by the success of Google [4].

To provide better search results, Google utilizes two forms of context analysis. First, it uses the text associated with a link to decide on attributes for the linked site. This text provides the context of both the creator of the linking site and the user who clicks on the link at that site. The more times that a particular word links to a site, the higher that word is ranked for that site. Second, Google uses the actions of a user after a search to decide what the user wanted from that search. For example, if a user clicks on the first four links of a given search, and then does not return, it is likely that the fourth link was the best match. This provides the user's context for those search terms; the user believes that those terms relate to

that particular site.

Unfortunately, Google's approach to indexing does not translate directly into the realm of file systems. Much of the information that Google relies on, such as links between pages, do not exist within a file system. Also, Google's query feedback mechanism relies on two properties: users are normally looking for the most popular sites when they perform a query, and they have a large user base that will repeat the same query many times. Unfortunately, neither of these properties are true in file systems: (1) users usually search for files that have not been accessed in a long time, because they usually remember where recently accessed files reside and access them directly, and (2) there is generally only a single user for each set of files; thus, it is unlikely that frequent queries will be generated for any given file.

3 Access-based Context Analysis

This section outlines two approaches to automatically gathering attributes when a file is created or accessed. These approaches use the context of the user's session at the time a file is accessed to assign attributes. The first uses application assistance, and the second uses existing user inputs.

Application assistance: Although most computers can provide a vast array of functionality, most people use their computer for a limited set of tasks. Most of these tasks are performed by a small set of applications, which in turn access and create most of the user's files. Modifying these applications to provide hints about the user's context could provide invaluable attribute information.

For example, if a user executes a web search for "asparagus" and downloads several pictures, it is likely that these are pictures of "asparagus." Similarly, if a user saves an email attachment and the subject of the email is "Re: Marketing report" then it is likely that the attachment is related to both "marketing" and "report."

Existing user input: Although most users are not willing to input additional information, they already are willing to choose a directory and name for the file. Each of the sub-directories along the path and the file name itself probably contain context information that can be used to assign attributes. For example, if the user stores a file in "~/papers/FS/Attribute-based/Semantic91.ps," then it is likely that they believe the file is a "paper" having to do with "FS," "attribute-based," and "semantic."

Like Google, an attribute-based file system can obtain information from user queries. If a user initially queries the system for "semantic file system" and chooses a file that only contains the attribute "semantic," then the additional terms "file" and "system" could be applied to that file. Also, if the possible matches are presented in the order that the system believes them to be most relevant, having the user choose files further into the list may be an indicator of success or failure. Also, as is done in some web search engines, a system could elicit feedback from the user after a query has completed, allowing them to indicate the success of the query using some sort of scale. Unfortunately, as mentioned above, individual files are likely to have few queries, reducing the amount of information available through this method.

4 Inter-file Relationships

Once relationships are established, attributes can be shared between related files. This helps to propagate attributes among individually hard-to-classify files. In conjunction with approaches that generate attributes (such as application assistance or content analysis), such propagation should categorize a much broader set of files. This section outlines two approaches to automatically gather inter-file relationships. The first approach leverages user access patterns, and the second approach examines content similarities between potentially related files.

User access patterns: As users access their files, the pattern of their accesses provides a set of temporal relationships between files. These relationships have previously been used to guide a variety of performance enhancements [12, 16, 23]. Another possible use of this information is to help propagate information between related files. For example, accessing "SemanticFS.ps" and "Gopal.ps" followed by updating "related.tex" may indicate a relationship between the three files. Subsequently, accessing "related.tex" and creating "WhyCantIFindMyFiles.ps" may indicate a transitive relationship.

Inter-file content analysis: Content analysis will continue to be an important part of automatically assigning attributes. In addition to existing per-file analysis techniques, our focus on creating context-based connections between files suggests another source of attributes: content-based relationships. For example, some current file systems use hashing to eliminate duplicate blocks within a file system [2, 18], or even locate similarities on non-block aligned boundaries [13, 17]. Such content overlap could also be used to identify related files, by treating files with large matching data sets as related.

Often, users (or the system [19]) will keep several slightly different versions of a file. Although these files generally contain differences, often the inherent information contained within does not change (e.g., a user may keep three instances of their resume, each focused for a different type of job application). This gives the sys-

tem two opportunities for content analysis. First, content comparison can identify related files. Second, by performing content analysis solely on the differences between versions, it may be possible to determine version-specific attributes, making it easier for users to locate individual version instances.

5 Initial Findings

This section discusses insights drawn from trace analysis of user activity.

5.1 Exploring Creation-time Attributes

Figure 1 shows two charts indicating the percentage of files created by different programs within a single user's home directory. This data was gathered from a trace of a single graduate student's "home" directory tree over a one month period. The first chart shows a breakdown of every file created within the directory tree. The second chart shows a breakdown of files explicitly organized by the user (rather than created and named by a program for itself) and believed to have some permanence (rather than being temporary or scratch files). This excludes things such as caches, logs, program configuration files, compiler output, and CVS source repositories, which are all organized by an external entity (generally the programs that create them).

Although a large number of files were created within this user's home directory, most of the files were organized by user-invoked programs rather than by the user themselves. Most of the user-organized files in the trace were created by three applications: a text editor/email program (emacs), a web-browser (mozilla), and document creation tools (latex). The others were created by various manual FS tools (e.g., "cp," "cat," etc.).

Examining these results suggests how a combination of the automated attribute assignment techniques described above can provide useful context information:

- The web-browser can generate hints for the files that it creates. For example, in this trace, the file "~/docs/online.pdf" was downloaded after doing a search for "SML Robert Harper" and clicking through Robert Harper's home page until the SML programming guide was located.

- Files created by text editors are generally accessed in conjunction with various other files, creating inter-file relationships. In this trace, several source code files were accessed in conjunction with a file named "~/class/814/homework2.tex," indicating that the files probably all related to "class," "814," and "homework."

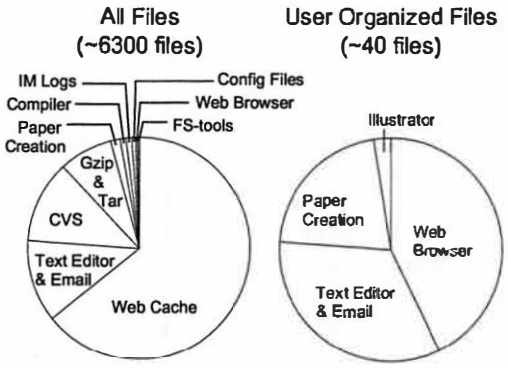


Figure 1: **Programs that create files.** Shows the programs that created files within a single graduate student home directory. The chart on the left shows a breakdown of every created file. Most of the files in this category are caches of either web pages or email, although archived source code (Tar & Gzip) and CVS repositories also figure in heavily. The chart on the right shows only those files explicitly organized by the student. These include those files downloaded from the web, hand edited files, files created by paper creation tools, and an image of a technical poster (Illustrator).

- Document creation tools like LaTeX take input from several different files and output a single postscript file (such as "homework2.ps"). This many-to-one relationship can be used to distill all of the input attributes into a smaller set of shared attributes that can be assigned to the output file. Also, these shared attributes could be passed back to any input files that do not have them.

- Illustrator (an image manipulation program) was used to create a poster outlining this work, importing text and images from a variety of related sources, resulting in a similar many-to-one relationship.

5.2 Exploring Inter-file Relationships

To further examine inter-file relationships, we created a simple tool to extract inter-file relationships from the trace. This tool tracks the last file access made by a program, and relates that file to the next file accessed. These relationships form groupings of related files.

Using this method, the tool successfully groups many files correctly (based on manual inspection by the owner). For example, a source tree was grouped with its resulting program output and backup tarballs, while a variety of unrelated source files were separated. Unfortunately, also grouped with the source tree were a variety of unrelated files (false positives). An examination of

the false positives showed that many were created by occasional use of *find* and *grep*. The graduate student in question uses *find* and *grep* to search by content for particular files. In a attribute-based naming system, *find* and *grep* would be replaced by an integrated searching system. This both removes the false positives, and could potentially improve accuracy using the feedback from user queries as described in Section 3.

6 Ongoing Challenges

Although our initial results are encouraging, there are still a large number of challenges beyond what has already been described. This section outlines some of these challenges, and initial ideas on how to approach them.

6.1 System Evaluation

One of the toughest research challenges faced when exploring automated attribute assignment is evaluating its accuracy. Although several groups have done automated file content analysis, little evaluation of the accuracy of these mechanisms has been reported. This is probably due to the difficulty of such an evaluation: what is "accurate?" More importantly, the true value of this kind of system is in helping users locate lost files, which is difficult to demonstrate without long-term deployment. Unfortunately, getting users to use such a system without first proving its value is difficult, resulting in a classic "Catch-22."

One possible approach is to feed a trace of user activity and application hints into the attribute assignment system and then compare its results to attribute assignment done by that same user. Unfortunately, this approach fails to account for user behavior. Although the user may initially categorize a file one way, they may later use it or look for it in another way. For example, the search terms they use a year after file creation may end up differing from their initial categorization.

6.2 Mechanisms

Although successfully assigning file attributes is one step in creating an attribute-based naming system, there are two other important aspects: the mechanism for storing attribute mappings and the user interface to the system. As mentioned in Section 2, several groups have looked at methods for storing attribute mappings. Until now, these methods have generally worked with a small number of attributes. By automatically identifying large numbers of attributes, two challenges arise. First, the existing methods may need to be extended to handle large numbers of attributes. Second, the system must identify the most relevant attributes for a file from the large set of associated attributes (i.e., weighting and false positive removal).

Several groups have also looked at the problem of user interfaces for attribute-based naming system. MyLifeBits [7] stores ⟨file, attribute⟩ pairings within a database, and provides a variety of file visualizations that help a user locate their files. Lifestreams [6, 20] provides a time-ordered stream of incoming information to the user, as well as a simple interface for filtering and sorting this information using a variety of attributes. Our work complements these and may provide useful insight into these two aspects of attribute-based naming.

6.3 User Context Switches

Context information has the potential to provide a large number of useful attributes. When a user switches context, however, the relationships created may be invalid. It would be helpful if the system could notice user context switches. One solution is user input, where the user indicates to the system what they are currently working on. If the user is not diligent, however, then the system may create more false positives than before. Another possibility is to infer user context switches from their actions. For example, switching to or from a particular application (e.g., the email browser) may consistently indicate a context switch.

7 Conclusions

As the data set associated with a user grows, organizing that information becomes more difficult. Although hierarchies have several useful aspects, they do not scale. A more flexible, attribute-based naming scheme is needed to effectively manage large personal data sets. This paper proposes automating attribute assignment using at-file-access context analysis and inter-file relationships. By obtaining many new attributes, these schemes should greatly increase the utility of attribute-based naming.

Acknowledgments

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[2] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. USENIX Windows Systems Symposium, pages 13–24. USENIX Association, 2000.

[3] C. M. Bowman, P. B. Danzig, U. Manber, and M. F. Schwartz. Scalable internet resource discovery: research problems and approaches. *Communications of the ACM*, 37(8):98–114, 1994.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[5] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/Gather: a cluster-based approach to browsing large document collections. ACM SIGIR International Conference on Research and Development in Information Retrieval, pages 318–329. ACM, 1992.

[6] S. Fertig, E. Freeman, and D. Gelernter. Lifestreams: an alternative to the desktop metaphor. ACM SIGCHI Conference, pages 410–411, 1996.

[7] J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong. MyLifeBits: fulfilling the Memex vision. ACM Multimedia, pages 235–238. ACM, 2002.

[8] D. Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann, 1998.

[9] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole Jr. Semantic file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 25(5):16–25, 13–16 October 1991.

[10] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. Symposium on Operating Systems Design and Implementation, pages 265–278. ACM, 1999.

[11] D. R. Hardy and M. F. Schwartz. Essence: a resource discovery system based on semantic file indexing. Winter USENIX Technical Conference, pages 361–373, 1993.

[12] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 31(5):264–275. ACM, 1997.

[13] J. MacDonald. *File system support for delta compression*. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[14] U. Manber and S. Wu. GLIMPSE: a tool to search through entire file systems. Winter USENIX Technical Conference, pages 23–32. USENIX Association, 1994.

[15] M. L. Mauldin. Retrieval performance in Ferret a conceptual information retrieval system. ACM SIGIR Conference on Research and Development in Information Retrieval, pages 347–355. ACM Press, 1991.

[16] G. Memik, M. Kandemir, and A. Choudhary. Exploiting inter-file access patterns using multi-collective I/O. Conference on File and Storage Technologies, pages 245–258. USENIX Association, 2002.

[17] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. ACM Symposium on Operating System Principles. Published as *Operating System Review*, 35(5):174–187. ACM, 2001.

[18] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies, pages 89–101. USENIX Association, 2002.

[19] D. S. Santry, M. J. Feeley, N. C. Hutchinson, R. W. Carton, J. Ofir, and A. C. Veitch. Deciding when to forget in the Elephant file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 33(5):110–123. ACM, 1999.

[20] Scopeware, http://www.scopeware.com/.

[21] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. International Conference on Distributed Computing Systems, pages 572–580, 1992.

[22] J. L. Steffen. Interactive examination of a C program with Cscope. Winter USENIX Technical Conference, pages 170–175. USENIX Association, 1985.

[23] S. Strange. *Analysis of long-term UNIX file access patterns for applications to automatic file migration strategies*. UCB/CSD–92–700. University of California Berkeley, Computer Science Department, August 1992.

Secure Data Replication over Untrusted Hosts

Bogdan C. Popescu
bpopescu@cs.vu.nl

Bruno Crispo
crispo@cs.vu.nl

Andrew S. Tanenbaum
ast@cs.vu.nl

Department of Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Data replication is a widely used technique for achieving fault tolerance and improved performance. With the advent of content delivery networks, it is becoming more and more frequent that data content is placed on hosts that are not directly controlled by the content owner, and because of this, security mechanisms to protect data integrity are necessary. In this paper we present a system architecture that allows arbitrary queries to be supported on data content replicated on untrusted servers. To prevent these servers from returning erroneous answers to client queries, we make use of a small number of trusted hosts that randomly check these answers and take corrective action whenever necessary. Additionally, our system employs an audit mechanism that guarantees that any untrusted server acting maliciously will eventually be detected and excluded from the system.

1 Introduction

Secure data replication on untrusted hosts has received a considerable amount of attention in the past few years. There are two generic mechanisms to handle this problem: state signing and state machine replication [16]. Solutions based on state signing [7, 2, 6, 11, 13, 3] can only support semi-static data content and restrictive, pre-defined types of queries. Furthermore, all these systems, except for the one described in [11], require that state updates are executed on trusted servers. On the other hand, systems based on state machine replication [4, 15, 10] allow untrusted servers handle the updates and support random queries, but require any particular operation to be executed multiple times (on different hosts), which greatly increases the amount of computing resources needed.

In this paper we present a system architecture that allows dynamic data replication with support for random queries, while avoiding much of the overhead associated with state machine replication. We are able to achieve this by providing only statistical guarantees on the correctness of any given query, combined with a background audit mechanism that detects false responses with a high degree of probability so corrective action can be taken. Our system is configurable, so it can easily provide 100% correctness and/or 100% false response detection, at the expense of operational performance.

Allowing erroneous behavior and taking corrective action only **after** an error has occured may seem a strange policy; however, our model is based on the assumption that byzantine failures from untrusted components of the system are rare, so the system can be optimized to give best performance in common case, which is when everything works correctly.

This paper is organized as follows: Section 2 introduces our system model, Section 3 describes the algorithms used to handle read and write operations on replicated data, Section 4 discusses several variants of our basic algorithms, and the operational scenarios where such variants may be appropriate, Section 5 reviews the related work in this area, and Section 6 concludes.

2 System Model

In this paper we consider a system model consisting of the following elements:

- The **data content**; this can be a database, the contents of a large Web site, or a file system. The data content needs to support both read and write operations; however, in our model we expect the number of reads to be at least an order of magnitude larger than the number of writes. The read operations can be very complex; they can request parts of the data content, but also the results of applying aggregation functions on this content. Taking the example of a file system, it should not only support operations of the type *read FileName*, but also operations of the type *grep Expression Path*.

- The **content owner**; this is one individual or organization which administers the content, and is in charge of setting an access control policy for it. For the purpose of this paper, we assume that data secrecy is not an issue, so the access control policy is only concerned with operations that modify the content.
- The **content key**; this is a public/private key pair associated with the data content. The content private key is known only by the content owner, while the content public key needs to be known by every client that wants to use the data. The latter can be accomplished by making this key part of the content identifier, as suggested in [5].
- The **master servers**; these are trusted hosts directly controlled by the content owner, each of them holding a copy of the data content. All the master servers in the system form the **master set**. There is a public/private key pair associated with each master server. The master servers' public keys are certified through digital certificates issued by the content owner (and signed with the content key). These certificates bind each server's contact address (IP address and port number) to its public key, and are stored in a public directory, indexed by content public key. Thus, by knowing the content public key and the address of the directory, any client can securely get the addresses and public keys of all the master servers replicating that content.
- The **slave servers**; they hold copies of the data content but are not directly controlled by the content owner, and because of this, they are only marginally trusted. They can be part of a content delivery network run by a separate organization, or managed by a number of cooperating, but mutually-suspicious institutions. There is a public/private key pair associated with each slave, and each master keeps track of the contact addresses and public keys of the slaves it has been assigned.
- The **clients**; they perform read/write operations on the data content. For a client to use the system, it first has to go through a setup phase, when it connects to exactly one master and one slave. First, the client queries the directory and selects one master (the closest one for example) to which it establishes a secure connection (using the master's certified public key). The master then sends the client the address and public key of one of its slaves (the one closest to the client for example) to which the client also establishes a secure connection. This concludes the setup phase; at this point the client can start issuing read/write requests - by sending them to either the master or the slave - according to the algorithm described in the next section.

3 Algorithm

In this section we present the algorithm used to handle read/write requests from clients. This algorithm guarantees that write requests are always processed correctly in some sequential order. However, in order to allow fast processing of read requests, only statistical guarantees are given for their correctness; this is based on our original assumption that byzantine failures from untrusted (slave) servers are infrequent.

Our algorithm requires the masters to be fully connected to each other through secure (e.g. cryptographically) communication links, and implement a reliable, total-ordering, broadcast protocol that can tolerate benign (non-malicious) server failures. The broadcast protocol itself is outside the scope of this paper; a good choice could be for example the protocol described in [8].

Using this reliable broadcast protocol, master servers ensure they always agree on the same sequential ordering for write requests. Through the same broadcast protocol, the masters also elect one of them to function as an **auditor**. The auditor checks (in the background) the correctness of computations performed by slaves, and takes corrective action when any of them is found acting maliciously.

Each master server is responsible for updating the servers in its slave set when the data content changes due to writes. This updating occurs only **after** the masters have comitted the write. The masters also periodically broadcast their slave list to the master set, so in the event of a master crash, the remaining ones will divide its slave set. This also entails that all the clients connected to the crashed server will have to go through the setup process again.

It is important to notice that a slave receives a state update only **after** that update has been comitted. The reason we have chosen this "lazy" state update algorithm, as opposed to having masters **and** slaves participate in the total ordering broadcast, is performance. Since only masters are trusted, a total ordering broadcast protocol including the slaves would have to be resistant to byzantine failures, and implementing such an algorithm over a WAN is extremely expensive. "Lazy" state updates make the write protocol much more efficient, but also weaken the consistency model since a client cannot be guaranteed that once his write is comitted it will be seen in all subsequent reads. We tackle this problem by introducing a special parameter, dubbed $max\_latency$ that bounds the inconsistency window for a given write operation: a client is guaranteed that once $max\_latency$

time has elapsed since comitting a write, no other client will accept a read that is not dependent on that write. It is worth stressing out that we do not guarantee that a write will propagate to all slaves in $max\_latency$ (this will violate the asynchronous nature of the WAN environment): there may be slaves for which it takes longer that $max\_latency$ to get a state update, but if they behave correctly they should stop handling user requests until they are back in sync (later we will show how to handle malicious slaves).

3.1 Write Protocol

Writes are executed only on trusted (master) servers. When a client wants to perform a write, it sends the request to its assigned master, which first checks whether the client is allowed to invoke such a request, and if this is the case, it broadcasts the request to the other servers in the master set. Upon the successful completion of this broadcast, each master executes the request and increments a special variable, dubbed here $content\_version$ (which is initialized zero when the content is created). In the end, all master servers hold updated, but still identical copies of the data content (because they have executed the same write) and have the same value for the $content\_version$ variable.

At this point, the slaves are updated: each master sends the update together with the signed and time-stamped and new value for the $content\_version$ variable to all its subordinates through a secure broadcast. In order to prevent race conditions, two write operations cannot be, time-wise, closer than $max\_latency$ to each other. This ensures that any reads on which the second write depends will take into account the first write. This obviously limits the number of write operations that can be executed in a given time, which is why we advocate our architecture only for applications where there is a high reads to writes ratio.

In order to satisfy the latency constraint, the master servers have to periodically broadcast "keep-alive" packets consisting of the signed and time-stamped value of the $content\_version$ variable to their server set, even when no writes occur. A slave can handle client requests only if the most recently receive "keep-alive" packet is less than $max\_latency$ old.

3.2 Basic Read Protocol

As mentioned, the replicated data content needs to support flexible query operations (reads); calculating the result of such a query can be a computationally very intensive task if it requires applying an aggregation function on the entire data content (a complex join for a database, or a *grep Expression Path* request in the case of a file system). Therefore, in order decrease the workload on the master servers and improve performance, read requests are handled by the slaves.

When a client wants to perform a read, it sends the request to its assigned slave server. The slave executes the request, and constructs a "pledge" packet which contains a copy of the request, the secure hash (SHA-1 [1]) of the result, and the latest time-stamped $content\_version$ value received from the master. After signing this "pledge" packet, the slave sends it to the client, together with the result of the query.

At the other end, the client first computes the secure hash of the query result, and makes sure it matches the hash in the "pledge" packet. Then, the client verifies the slave's signature on the "pledge" packet. Finally, the client makes sure the time-stamp is not older than $max\_latency$. If all these conditions are met, the client accepts the answer, otherwise it rejects it.

Because of the asynchronous nature of the network connection between the client and the slave, it is possible that a result that was "fresh" when sent by the slave, becomes stale by the time it reaches the client. In such a situation, the client has to drop the answer and try the query again. By carefully selecting the value for $max\_latency$, and the frequency masters send "keep-alive" packets, the probability of such events occuring can be reduced. However, clients with very slow or unreliable network connections may never be able to get fresh-enough responses. One possible way to accommodate such clients is to relax the consistency model and allow the $max\_latency$ to be set by the clients themselves. In this case, clients with fast network connections can set high "freshness" requirements, while clients with slow connections can settle with more modest expectations.

The main vulnerability of this basic read protocol is that a malicious slave can return wrong answers to client requests. To protect against such malicious slaves, we employ two techniques - probabilistic checking and auditing which will be described in the next two sections.

3.3 Probabilistic Checking

For each read request, there is a certain probability that a client will send the same request to a master and compare the slave and master results. This "double-check" probability is a system parameter - it should be small enough so it does not excessively increase the workload on the masters, but large enough so it guarantees that a malicious slave is caught "red-handed" quickly.

As described in the previous section, for every read it handles, a slave has to sign a "pledge" packet that contains a copy of the request, the content version time-stamped by the master and the secure hash of the result (as computed by the slave). Should the slave act maliciously and return an incorrect answer, the "pledge"

packet becomes an irrefutable proof of its dishonesty. Once a slave server is proven malicious it can be excluded from the system so it can do no further harm.

It is important to notice that the way the "pledge" packets are constructed makes impossible for a client to "frame" an innocent slave server - unless that client is able to fake the slave's digital signature. The only harm a client can do is to abuse its "double-check" quota (by double-checking all the slave responses instead of a small fraction of them). However, by keeping track on the number of double-check requests it receives from each of its clients, a master can identify statistically anomalous client behavior, which most likely indicates a "greedy" client. The master can then enforce fair play by simply ignoring a large fraction of the double-check requests coming from clients suspected to be greedy.

3.4 Auditing

Besides relying on the probabilistic checker to detect wrong answers, our system also employs an auditing mechanism that ensures that even if a malicious slave manages to return an erroneous result to a client, that slave will eventually get caught and excluded from the system.

The auditing mechanism works as follows: after the client accepts the result from the slave, and given that the client decides not to double-check that result, it still forwards the slave's "pledge" packet to the special auditor server.

The auditor is the only trusted server that does not have a slave set, and therefore does not handle any double-checking requests from clients; its only duty is to check the validity of "pledge" packets, by re-executing the read request in the packet and comparing the secure hash of the result to the hash in the packet. The auditor is allowed to lag behind when executing write requests; it executes a write only after it has audited all the read requests for the *content_version* that precedes that write. In fact, in order to take into account possible network delays, the auditor can move to a new *content_version* only after a sufficiently large time interval (more than *max_latency*) has elapsed since the rest of the trusted servers have moved to that same *content_version*. This ensures that at that point no client will accept any more read results for the previous *content_version*. Here we assume clients accept read results only after they have forwarded the corresponding pledges to the auditor.

The auditor has several advantages over the slaves it has to verify, which allow it to achieve a much higher throughput when (re)executing read operations: first the auditor does not have to produce digital signatures (slaves on the other hand have to digitally sign a "pledge" packet for every client request they execute). Second, the auditor does not have to send any answers back to the clients. Third, since the auditor knows in advance all the operations it has to re-execute, it can, for certain types of applications (for example databases), employ query optimization mechanisms (cache results in the simplest case). Finally, because the auditor needs not to worry about client latency, it can spread its work so it minimizes idle time. Assuming that read requests show daily peak patterns (few requests at 3AM in the night for example), it is possible that the auditor will seriously lag behind during peak hours, but catch up during the night. However, it is essential that in the long run the auditor is able to keep up with the amount of reads it has to verify. If the auditor is over-used, the solution is to either add extra auditors, or weaken the security guarantees by verifying only a randomly chosen fraction of all reads.

3.5 Taking Corrective Action

In this section we discuss what happens when one of the slave servers is caught "red-handed," either as a result of a client double-checking a read result with a master (immediate discovery), or during the audit process (delayed discovery).

In the case of immediate discovery, the client forwards the incriminating "pledge" packet to the master. At this point, the master contacts all the clients connected to the (now provably malicious) slave, informs them the slave has been excluded from the system, and assigns each of them to a new slave server. Finally, the client that has made the discovery connects to its newly assigned slave and issues the same read request again.

In the case of delayed discovery, the situation is more complex, since at least one client has already accepted an incorrect answer. In some applications, the harm may be undone, by rolling back the client to the state before that particular read. In any case, the malicious slave needs to be excluded from the system so it can do no further damage. To this extent, the auditor sends the incriminating "pledge" packet to the master in charge of the slave that has signed it. The master then contacts all the interested clients (clients connected to the malicious slave) informing them of the problem and assigns them to new slave servers.

What happens after a malicious server has been excluded from the system is dependent on the administrative relations between that server and the content owner. If a formal content hosting contract exists, the matter can be further taken to courts (given that the incriminating "pledge" packet can be used as evidence). Another possibility is that the slave server is not inherently malicious, but is has been the victim of an attack (the trusted servers are supposedly administered much more carefully, so they cannot be easily hacked), in which case,

after recovering it to a safe state, it can be brought back to use.

4 Discussion

The algorithm described is based on the optimistic assumption that byzantine failures are rare, allowing the system to be optimized for the common case, where untrusted components work correctly, even if there is a danger that incorrect results may be passed to clients. The probabilistic checking and auditing mechanisms guarantee that components acting maliciously will eventually be identified and excluded from the system.

There may be situations when stronger correctness guarantees are required. We outline a number of variants of our algorithm to handle this situation:

One variant is to give clients the option to differentiate between "normal" and "security sensitive" reads. The latter ones are then to be executed only by the trusted servers (which guarantees that clients always get correct results), while normal (non-sensitive) operations are carried out as described above. This variant allows us to provide 100% correctness guarantees for sensitive operations, at the expense of putting extra load on the trusted components. A further refinement of this scheme assigns even more security levels for read operations and sets the double-check probability based on the read's security level. Thus for the least sensitive operations the probability can be set very low, while for the very sensitive it can be set to 1 (which means "execute only on trusted hosts").

Another possibility is to send the same read request to more than one untrusted server. If all the answers are identical, the client proceeds as in the original algorithm - double-check with the master (with a small probability) and send the "pledge" packets, to the auditor. If not all answers match, the client automatically double-checks, since at least one of the slaves has to be malicious. This approach is similar to what is proposed in [4], and has the advantage that a number of malicious slaves would have to collude in order to pass an incorrect answer. The disadvantage is that more computing resources are needed in order to handle the same request, but these resources need not be trusted, and may therefore be easier to come by.

5 Related Work

There are two generic mechanisms for securely replicating data over untrusted hosts: state signing and state machine replication.

With state signing, the data content is divided into small (disjunct) subsets which are signed with a content private key. Clients then retrieve data from untrusted storage and verify its integrity using the content public key (assumed to be known a-priori). In order to minimize the number of digital signatures, some form of hash-tree authentication [12] is normally used in this context.

Work that falls into this category includes [7, 11, 13, 3] which apply this technique to distributed file systems, [2] which applies it to free software distribution, [6] which applies it to signing XML documents, [14] which applies it to digital certificate revocation and [9] which applies it on building a trusted database on untrusted storage. The main limitation for all these systems is that dynamic queries on the data need to be executed on trusted hosts. This requires the trusted host to first retrieve all data relevant to the query from untrusted storage, verify it, and then perform the operation.

With state machine replication [16], the idea is to execute the same operation on a number of untrusted hosts (quorum), and accept the result only when a majority of these hosts agree upon it. In this way, malicious hosts need to collude in order to pass an erroneous result; by requiring a large quorum size, the system can offer very strong security guarantees. Work in this area includes [4, 15, 10, 17]. The problem with this approach is that it greatly increases the amount of computing resources needed for handling a given request. Additionaly, the request latency is dictated by the slowest server in the quorum group.

The scheme we describe in this paper allows dynamic queries to be handled by untrusted servers, while avoiding most of the overhead associated with state machine replication. We are able to achieve this by providing only statistical guarantees on the correctness of any given query. However, our background auditing mechanism ensures that malicious servers are eventually detected, so they can be handled appropiately (either brought back to a safe state, or removed from the system).

6 Limitations and Future Work

One of the possible usage scenarios for the system architecture described in this paper is in the area of content delivery networks (CDNs), used for replicating semi-static Web content such as product catalogues for e-commerce, or academic, medical and legal databases. One possibility is having the organization that owns the data content to provide the master servers, while the CDN provides the slaves. Yet another possibility is to have the CDN itself divide its servers in a trusted core and a much larger set of outsourced and thus less trusted support servers. This scenario seems particularly realistic given the fact that most CDNs physically host most of their servers with Internet service providers and only remotely administer them.

The work presented in this paper is based on the fundamental assumption that byzantine failures are rare

events, so applications can be optimized to work efficiently in the common case - when everything works correctly. This assumption is also the major limitation of our approach as it cannot be used (or at least is not efficiently) in scenarios when 100% security guarantees are required. However, looking at the current state of the Internet (the vast majority of WWW traffic is not encrypted, and even secure DNS is slow in gaining acceptance) it seems there are numerous applications where people can do well even without strong security guarantees.

The other limitation of our approach is that there is a certain latency for propagating writes, and in order to avoid race conditions we need to limit the frequency of such operations. As a result, the architecture described in this paper is appropriate for applications with a high reads to writes ratio. CDNs used for replicating slowly changing Web content, as well as academic, legal or medical databases clearly fall in this category. On the other hand, it would be impractical to use this architecture for disseminating data that changes rapidly and requires tight freshness guarantees, such as live stock quotes.

References

[1] Secure Hash Standard. FIPS 180-1, Secure Hash Standard, NIST, US Dept. of Commerce, Washington D. C. April 1995.

[2] A. Bakker and M. van Steen and A. Tanenbaum. A law-abiding peer-to-peer network for free-software distribution. In *Proc. Int'l Symp. on Network Computing and Applications*, Cambridge, MA, Feb. 2002. IEEE.

[3] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[5] D. Mazieres and F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *In Proc. 8th ACM SIGOPS European Workshop*, pages 118–125. ACM, 1998.

[6] P. T. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of XML documents. In *ACM Conf. on Computer and Communications Security*, pages 136–145, 2001.

[7] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[8] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, 1989.

[9] U. Maheshwari, R. Vingralek, and B. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI: 4th Symposium on Operating Systems Design and Implementation*, pages 135–150. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 2000.

[10] D. Malkhi and M. K. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, pages 51–58. IEEE, 1998.

[11] D. Mazieres and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. of the 21st Annual Symposium on Principles of Distributed Computing*, pages 108–117, Monterey, CA, 2002. ACM.

[12] R. C. Merkle. A certified digital signature. In *Proc. Crypto '89*, LNCS 435, pages 234–246. Springer-Verlag, 1989.

[13] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong security for network-attached storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Jan. 2002.

[14] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium*, Jan 1998.

[15] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Lecture Notes in Computer Science*, volume 938, pages 99–110. Springer-Verlag, Berlin Germany, 1995.

[16] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[17] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

Palimpsest: Soft-Capacity Storage for Planetary-Scale Services

Timothy Roscoe
Intel Research at Berkeley
2150 Shattuck Avenue, Suite 1300
Berkeley, CA, 94704, USA
troscoe@intel-research.net

Steven Hand
University of Cambridge Computer Laboratory
15 J.J. Thompson Avenue
Cambridge CB3 0FD, UK
steven.hand@cl.cam.ac.uk

Abstract

Distributed writable storage systems typically provide NFS-like semantics and unbounded persistence for files. We claim that for planetary-scale distributed services such facilities are unnecessary and impose an unwanted overhead in complexity and ease of management. Furthermore, wide-area services have requirements not met by existing solutions, in particular capacity management and a realistic model for billing and charging.

We argue for ephemeral storage systems which meet these requirements, and present Palimpsest, an early example being deployed on PlanetLab. Palimpsest is small and simple, yet provides soft capacity, congestion-based pricing, automatic reclamation of space, and a security model suitable for a shared storage facility for wide-area services. It does not "fill up" in the way that an ordinary filing system does. In fact, Palimpsest does not arbitrate resources—storage blocks—between clients at all.

1 The Need for Ephemeral Storage

In this paper we address the distributed storage needs of wide-area services designed to run over the kind of shared, planetary-scale computational infrastructure envisioned by the PlanetLab[12], Xenoservers[14], WebOS[18] and Denali[20] projects. So-called "persistent data" in such services consists of the (relatively static) code and data which comprises the service, configuration information, output in the form of log files and transaction records, and system state such as checkpoints and scoreboards. This is a different application area than that addressed by filing systems intended for human users.

We observe that *almost all* this data is either ephemeral, or else a current copy of data which is or will shortly be held or archived elsewhere. Nevertheless, it must be stored with high availability for some period of time. For example, checkpoint state need only be kept until the next checkpoint has been performed, but before then is of high value. Similarly, transaction records must be periodically copied offline for processing, but until this happens it should be hard for a system failure or malicious agent to delete the data or obtain unauthorized access to it.

Current practice is to store such data on a disk directly attached to its node, or use a distributed storage service. The former fails to provide availability in the event of a node failure, while the conventional examples of the latter implement unnecessarily strict semantics and fail to address the resource issues involved in sharing the storage resources between many independent wide-area services.

Similarly, file system security requirements for such services differ from traditional time-sharing systems. Often it should be hard to delete the data or for unauthorized entities to access it, but there are no requirements for complex access control arrangements. Likewise, since the "users" of the storage service will be software services themselves, the human factors issues associated with key storage are largely obviated.

On the other hand, a shared, widely distributed environment imposes other requirements and in particular the need to deal gracefully with a shortage of capacity. While it is true that storage in conventional environments (PCs, workstations, LANs) is generally underutilized, we point out that in these cases the storage is administered by a single organization who has an interest in limiting usage or buying new disks when needed. This is not at all the case in a shared, distributed environment where multiple services must be given incentives to release storage resources and share storage space among each other, particularly when that storage is provided by a third party. Furthermore, wide-area storage services "in the wild" must be robust in the face of denial-of-service attacks, including those that use up storage space.

Distributed filing systems which aim at NFS-like semantics or something similar (e.g. [11, 15]) do not meet these requirements, and employ considerable complexity and incur considerable overhead in providing behavior which, while essential in a LAN environment with

human users, is inappropriate in our scenario. Furthermore, they do not address the capacity and denial-of-service requirements mentioned above. Some recent wide-area file systems[5, 17] have moved away from an NFS-like model to, for instance, immutable and read-only file service, neither of which is a requirement in our case.

In the rest of this paper, we give an overview of Palimpsest[1], followed by more detail on the central algorithms and architecture. Along the way we try to point out engineering trade-offs and other areas for further investigation. We then discuss charging for storage based on congestion pricing techniques, and end with comparison of Palimpsest to other wide-area file systems.

2 An Overview of Palimpsest

Palimpsest is not a mainstream, traditional distributed filing system. It does not implement NFS semantics, does not provide unbounded persistence of data, and implements a security model different from that traditionally derived from multi-user timesharing environments. It does not "fill up" in the way that an ordinary filing system does. In fact, Palimpsest does not arbitrate resources—storage blocks—between clients at all.

Instead, Palimpsest delivers storage with true, decentralized soft-capacity (clients get a share of the overall storage resource based on the current total number of users, and how much each is prepared to pay), and removes the burden of garbage collection and resource arbitration from storage service providers. Data stored by clients in Palimpsest is secure, and persists with high availability for a limited period of time which is dynamically changing but nevertheless predictable by clients. In addition, Palimpsest provides a novel, financially-viable charging model analogous to congestion pricing in networks and highly appropriate for anonymous microbilling, further reducing administrative overhead.

In Palimpsest, writers store data by writing erasure-coded fragments of data chunks into an approximation of a distributed FIFO queue, implemented by a network of block stores. Over time, as more writes come in, fragments move to the end of the queue and are discarded. A reader requests fragments from the nearest set of storage nodes according to some desirable metric, such as latency. The combination of a distributed store and erasure coding provides high availability.

Storage nodes don't care who writes blocks, but instead charge for each operation. Billing can thus be performed using digital cash, and Palimpsest doesn't need to retain any metadata related to user identity at all.

Each storage node maintains a value called its *time constant*, which is a measure of how fast new data is being written to the store and old data discarded. By sampling the time constant from a series of storage nodes (an operation which can be piggybacked on reads and writes), a writer can obtain good estimates of how long its data will persist, even though this value changes over time.

Consequently, Palimpsest does not "fill up"—in normal use, it's always full. Instead, as the load (in writes) increases, the time over which data persists decreases. The problem of space allocation or reclamation thus never arises. Instead, Palimpsest introduces a new trade-off between storage capacity and data persistence. This trade-off allows the application of techniques originally developed for congestion pricing of network bandwidth to be applied to storage, a concept we explore below.

3 The Design of Palimpsest

The basic functionality provided by Palimpsest clients is the storage or retrieval of a *chunk* of data. This is a variable length sequence of bytes up to some maximum value (128kB in our current prototype). To store a chunk, it is first erasure-coded (using Rabin's Information Dispersal Algorithm[13]) into a set of m fixed-size fragments (currently 8kB), any n of which suffice to reconstruct the chunk. n is simply the chunk size divided by the fragment size; m can be chosen by the client to give the desired dispersion factor — the trade-off here is between resilience to the loss of fragments versus storage cost and network bandwidth.

The fragments are then authentically encrypted (using the AES in Offset Codebook Mode[16]) to produce a set of fixed size blocks which are then stored at a pseudo-random sequence of locations within a 256-bit virtual address space; a distributed hash table maps the 256-bit virtual block numbers to block identifiers on one of a peer-to-peer network of block stores. To recover the chunk, the pseudo-random sequence is reconstructed and requests made for sufficiently many blocks to reconstitute the chunk.

The pseudo-random sequence is generated by successive encryption under our secret key of a per-chunk initial value, and so provide statistical load-balancing over the storage nodes in the system. It also makes it hard for an adversary to selectively overwrite a file since the block identifiers are unpredictable without the key. Data is already protected against tampering or unauthorized reading by the encryption and MAC. A blanket attempt to overwrite all data is rendered prohibitively expensive by the fact that Palimpsest charges when a block write transaction takes place (see below).

The use of the IDA also means that Palimpsest will tolerate the loss of a fraction of block servers, and also

the loss of individual write operations or the failure of a fraction of read operations.

Block servers hold a FIFO queue of fragments which is also indexed by the 256-bit block identifier. When a write request for a block id b is received, b is looked up in the index. If an existing block with id b is found, it is discarded and the new block added to the tail. Otherwise, the head of the queue is discarded and the new block added to the tail. Block stores thus perform no arbitration of resources — there is no "free list", neither is the block store interested in the identity of the writer of any block.

Block servers also keep track of the arrival rate of block write requests, expressed as a fraction of the size of the server's store[2]. This number is the *time constant* of the server, and is sampled by clients as described below.

This basic chunk storage is the only facility provided by Palimpsest. Filing systems can be implemented above this using standard techniques, but the details of this are entirely up to the application.

3.1 Persistence and the Time Constant

As new blocks are written to block stores, "older" blocks are eventually discarded. This will ultimately make any given chunk irretrievable; if this is not desired, clients need to take steps to ensure their data remains live.

This persistence can be attained by *refreshing* chunks from time to time. To aid clients in deciding when to perform refresh operations, each block store keeps track of its *time constant*, τ: the reciprocal of the rate at which writes are arriving at the store as a fraction of the total size of the store.

If our distributed hash table adequately approximates a uniform distribution, each block store will tend to have the same time constant. If the global load on Palimpsest increases, or several block stores are removed from the system, the time constants of remaining stores will decrease. Conversely, if new capacity (in the form of block stores) is added to the system, or the total write load to Palimpsest decreases, the time constants will increase.

Each read or write operation for which a response is obtained includes the value of τ at the block store when the response was sent. Clients use these values to maintain an exponentially weighted estimate of system τ, τ_s. This quantity can then be used to determine the likely expiration of particular blocks, which may then be used to drive the refresh strategy.

The most straightforward refresh strategy would simply write a chunk, derive an estimated τ_s based on the responses, wait almost that long, and then rewrite the chunk. To handle unexpected changes in the value τ after the chunk has been written, clients can periodically sample the block stores. Since blocks are moving through the FIFO queues in the block, τ_s and consequently the refresh rate can be adjusted in a timely fashion before data is lost.

More sophisticated refresh strategies are an area for research, although we expect at least some will be application-specific.

We note that the value of τ acts as a single, simple, system-wide metric by which administrators may decide to add more storage capacity. This contrasts with the complex decision typically faced by operators of distributed storage systems.

3.2 Concurrent Reads and Writes

Reconstituting a chunk of data stored in Palimpsest introduces some subtle challenges in the presence of concurrent reads and writes to the same chunk. Palimpsest addresses this by guaranteeing that reads of a chunk will always return a consistent chunk if one exists at the time.

When a Palimpsest client requests a block during chunk retrieval, three outcomes are possible:

1. the block server returns the "correct" block (viz. the one which was stored as part of the chunk);
2. the server is unavailable, or does not hold a block with the requested identifier (resulting in an error response or a timeout);
3. a block is returned, but it is not part of the chunk in question.

The third case is not entirely detected by the use of the MAC, since if a writer updates the chunk in place, using the same pseudo-random sequence to write the new blocks, it becomes impossible to tell which *version* of a chunk each block is from. This in turn leads to a situation where a chunk cannot be reconstituted in reasonable time because, although enough valid blocks are theoretically available, a small number are from an older version and "poison" the process of reversing the erasure code.

We could address this problem by attempting to recombine all combinations of fragments which pass the MAC check until we get a reconstituted chunk which passes some other, end-to-end integrity check. Apart from being inelegant, this becomes computationally impractical even with modest chunk sizes and redundancy in coding.

Another approach is to use a different sequence of block ids for each new version of the chunk. This approach was used in the Mnemosyne system due to its desirable steganographic properties, and while effective for small numbers of highly secure files, it has a major disadvantage for a typical Palimpsest scenario: it requires a

piece of metadata (the initial hash value of the "current" version) to be held externally to the chunk, but which changes whenever the chunk is updated. This causes a "cascading" effect when one tries to build a hierarchical filing system above Palimpsest's chunk storage: updating a file requires the file's metadata to change, which causes an update to an inode or directory chunk, which causes *its* metadata to change, and so on up to the root.

Consequently, Palimpsest adopts a third approach: including a version identifier in each fragment. This allows a valid chunk to be reconstituted if enough fragments can be retrieved, but a new version of the chunk to be written to the same block sequence without modifying the chunk metadata. Even if the chunk length changes and an inode also needs to be updated, this technique prevents updates cascading up the directory tree.

Version identifiers for chunks need not be sequential, or indeed monotonic. Palimpsest uses the upper 64 bits of the AES-OCB nonce (which is stored with the block) to denote the version of the chunk, and uses voting to decide which fragments should be used to reconstitute the chunk. To retrieve a chunk, some number of fragments is requested and histogrammed according to version identifier. If any histogram bin contains enough fragments to reconstitute the chunk, the algorithm succeeds. If no more fragments can be requested, it fails. Otherwise more fragments are requested.

The derivation of good schedules for retrieval of fragments (how many to request at each stage, how to use network knowledge to choose which fragments to request, etc.) is a field rich in possible trade-offs, and is an open area of research.

How to behave when the above algorithm fails is application specific. For example, if the reader believes the data to be periodically updated, it can immediately retry since the failure may be due to a concurrent write.

3.3 Charging and Billing

As we argue in Section 1, storage space is a *scarce* resource in planetary-scale systems: the popular wisdom that storage is "free" only applies in local or small-scale systems. Palimpsest must therefore induce clients to moderate their use of storage capacity without any knowledge of either the global set of clients, or the amount of storage each is using.

We achieve this by using a micropayment scheme in which every `read` and `write` operation must be accompanied by a digital token. A two-tier charging model is used since while `read` operations have only a direct cost to the provider (that of locating and serving the relevant block), `write` operations have both direct and indirect costs: in the steady-state, each `write` will displace exactly one existing data block and with some probability p will render a certain file inaccessible. Hence `write` operations warrant a higher charge than `reads`.

Since a charge is imposed with each use of the system, and since increased use by a user i adversely affects all other users, we can make use of control algorithms based on *congestion pricing* [9]. In general this means that we ascribe to each user i a utility function $u_i(x_i, Y)$ where x_i is the number of `write` operations performed by user i and Y is the "congestion" of the entire system. We need u_i to be a differentiable concave function of x_i and a decreasing concave function of Y; this corresponds intuitively to notion that storage in Palimpsest is "elastic": users always gain benefit from writing more frequently, and from less competition from other users.

Of course the incentive to increase one's write-rate is balanced by the cost of paying for it. In traditional congestion pricing schemes (designed for regulating use of network resources), a spot-price is advertised at any point in time; this price is computed so that an increase in demand is balanced by the marginal cost of increasing the resource. In Palimpsest, rather than explicitly increasing the price, we simply advertise the new time constant – for a given level of risk, this will result in a rational client modifying their refresh rate so as maintain the expected data lifetime they require. We believe this improves on the scheme proposed in e.g. [8] since we explicitly model the variation in τ and predict its future values rather than relying on instantaneous feedback.

As an alternative, a storage service can advertise a guaranteed time constant to its clients. It can then use the time constant measurements made by the block stores to perform the analogue of network traffic engineering: working out when to provision extra capacity, and/or change its advertised service. Note that, under this model, denial of service attacks in Palimpsest are actually an expansion opportunity for the storage provider: maintaining an appropriate τ can be achieved by increasing the storage resources commensurate with the additional revenue obtained.

As in the case with network congestion pricing, clients which require strong availability guarantees (or, equivalently, predictable prices) can deal with intermediaries who charge a premium in exchange for taking on the risk that a sudden burst of activity will force more refreshes than expected. The provision of strong guarantees above Palimpset then becomes a futures market.

3.4 Status

Palimpsest is under active development, and we hope to deploy it as a service on PlanetLab in the near future. The Palimpsest client is implemented in C, while the block server is written in Java and currently uses Tapestry as its DHT.

4 Related Work

Palimpsest shares some philosophy with the Internet Backplane Protocol[1], in that the conventional assumption of unbounded duration of storage is discarded. Palimpsest supports similar classes of application to IBP while pushing the argument much further: all storage is "soft capacity", with no *a priori* guarantees on the duration of persistence. In this way, storage services avoid performing any metadata management or resource arbitration, resulting in substantially simpler implementations. Resource management is pushed out to end systems by means of transaction-based charging and periodic sampling of the store's time constant.

The notion of having no explicit delete operation and relying on storage being "reclaimed" was perhaps first observed in the Cambridge File System[2]. Palimpsest shares the goal of avoiding reliance on users to remove files, rather putting the onus on them to refresh what they wish to retain. We avoid the need for an explicit asynchronous garbage collector by simply allowing incoming `write` operations to displace previous blocks.

A number of wide-area storage systems have recently been developed in the peer-to-peer community. Early systems supported read-only operation (e.g. CFS[5], Past[17]) though with high availability. As with distributed publishing systems (e.g. Freenet[4], Free Haven[6] or Publius[19]) we believe these systems complement rather than compete with Palimpsest. Ironically, Palimpsest's ephemeral storage service shares several implementation techniques (such as the use of erasure codes) with very long-term archival schemes like Intermemory[3].

More recently various efforts at Internet-scale read-write storage systems have emerged including Pasta[10], Oceanstore[15], and Ivy[11]. All are far more involved than Palimpsest, with schemes for space allocation or reclamation being either complex (in the first two cases) or non-existent. Palimpsest by contrast incorporates support for storage management at its core but without requiring intricate algorithms or centralized policies.

Some of the ideas in Palimpsest were inspired by the Mnemosyne steganographic filing system[7], though there are notable differences, since Palimpsest is targeted at widely distributed Internet services rather than individual security-conscious users. Palimpsest implements an approximate distributed FIFO queue rather than the random virtual block store in Mnemosyne, which increases both the effective capacity of the block store and predictability of chunk lifetimes. Palimpsest's use of charging, choice of encoding scheme, and selection of block identifiers also reflects its goal of providing a useful facility to planetary-scale services.

5 Conclusion

We have argued that emerging wide-area, planetary-scale services have different requirements of distributed storage than the existing models based on human users of time-sharing systems, or network distribution of read-only content. Highly general security policies and unbounded duration of file persistence come with considerable complexity and are largely not required in this area. Furthermore, current systems lack effective mechanisms for allocating scarce storage space among multiple competing users, and a viable economic model for resource provisioning.

We claim there is a role here for storage services which offer bounded duration of files, but provide high availability and security during that time, combined with soft capacity and a congestion-based charging model. Such a system is Palimpsest, which we have described and are in the process of deploying on the PlanetLab infrastructure. The design of Palimpsest allows a number of interesting design choices and lends itself to a lightweight, flexible and secure implementation.

Notes

[1] A palimpsest is a manuscript on which an earlier text has been effaced and the vellum or parchment reused for another.

[2] Currently, all block servers have the same size store, a decision well-suited to the current PlanetLab hardware configurations. We propose the use of virtual servers as in CFS [5] to allow greater storage capacity on some physical nodes.

References

[1] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *Proc. of ACM SIGCOMM 2002, Pittsburgh, PA.*, August 2002.

[2] A. Birrell and R. Needham. A universal file server. *IEEE Transactions on Software Engineering*, 5(6):450–453, September 1980.

[3] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries, Berkeley, California*, Aug. 1999.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[5] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles, Banff, Canada.*, October 2001.

[6] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, July 2000.

[7] S. Hand and T. Roscoe. Mnemosyne: Peer-to-Peer Steganographic Storage. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[8] P. Key and D. McAuley. Differential QoS and Pricing in Networks: where flow-control meets game theory. *IEE Proc. Software*, 146(2), March 1999.

[9] J. K. Mackie-Mason and H. R. Varian. Pricing Congestible Network Resources. *IEEE Journal of Selected Areas in Communications*, 13(7):1141–1149, September 1995.

[10] T. D. Moreton, I. A. Pratt, and T. L. Harris. Storage, Mutability and Naming in *Pasta*. In *Proc. of the International Workshop on Peer-to-Peer Computing at Networking 2002, Pisa, Italy.*, May 2002.

[11] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation, Boston, MA.*, December 2002.

[12] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of the 1st Workshop on Hot Topics in Networks (HotNets-1), Princeton, New Jersey, USA*, October 2002.

[13] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Communications of the ACM*, 36(2):335–348, April 1989.

[14] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accountable execution of untrusted programs. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HOTOS-VII)*, pages 136–141, 1999.

[15] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The Oceanstore Prototype. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies, San Francisco, CA.*, March 2003.

[16] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)*. ACM Press, August 2001.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large scale persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating Systems Principles, Banff, Canada.*, October 2001.

[18] A. Vahadat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proc. of the Seventh International Symposium on High Performance Distributed Computing, Chicago, IL*, July 1998.

[19] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceeding of the 9th USENIX Security Symposium*, pages 59–72, August 2000.

[20] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation, Boston, MA.*, December 2002.

Certifying Program Execution with Secure Processors

Benjie Chen Robert Morris
MIT Laboratory for Computer Science
{*benjie,rtm*}*@lcs.mit.edu*

Abstract

Cerium is a trusted computing architecture that protects a program's execution from being tampered while the program is running. Cerium uses a physically tamper-resistant CPU and a μ-kernel to protect programs from each other and from hardware attacks. The μ-kernel partitions programs into separate address spaces, and the CPU applies memory protection to ensure that programs can only use their own data; the CPU traps to the μ-kernel when loading or evicting a cache line, and the μ-kernel cryptographically authenticates and copy-protects each program's instructions and data when they are stored in the untrusted off-chip DRAM. The Cerium CPU signs certificates that securely identify the CPU and its manufacturer, the BIOS and boot loader, the μ-kernel, the running program, and any data the program wants signed. These certificates tell a user what program executed and what hardware and software environment surrounded the program, which are key facts in deciding whether to trust a program's output.

1 Introduction

Although research on the use of tamper-resistant hardware has been in progress for nearly 15 years [10, 11, 12, 7, 5], public concerns for issues such as copy protection and secure remote execution and the recent push in commodity secure hardware [2] suggest that the benefits of using secure hardware is now exceeding its overhead in complexity, performance, and cost. This paper describes a trusted computing architecture, Cerium, that uses a secure processor to protect a program's execution, so that a user can detect tampering of the program's instructions, data, and control-flow while the program is running.

This paper considers the following computation model. A user runs a program on a computer outside the user's control. The computer runs the program and presents the user with an output. The user wants to know if the output is in fact produced by an un-tampered execution of the user's program. We call this computation model *tamper-evident execution*. Tamper-evident execution enables many new useful applications. For example, a project that depends on distributed computation, such as SETI@home [1], can use tamper-evident execution to check that results returned by participants are produced by the appropriate SETI@home software.

The goal of Cerium is to support tamper-evident execution while facing strong adversaries. At the user level, Cerium should expose malicious users forging results of other users' programs without running them. At the system level, Cerium should expose buggy operating systems that allow malicious programs to modify the instructions and data of other programs. At the hardware level, Cerium should detect hardware attacks that tamper with a program's data while they are stored in memory, such as attacks on the DRAM or memory bus. Such strong adversaries prevent us from using software only techniques (e.g. Palladium [3] and TCPA [2]) to implement tamper-evident execution.

Cerium is designed to be open and flexible. Cerium does not limit which operating system or programs can run on a computer. Instead, Cerium tells a user what program executed and what hardware and software environment surrounded the program, so the user can decide whether to trust the program's output. This is in contrast to a more controlled and restrictive approach taken by some related systems [12, 7]. The IBM 4758 system [7], for example, provides a secure computing platform by allowing only operating systems and programs from trusted entities to run inside a secure co-processor. The co-processor establishes trust with a new entity (e.g. a bank) if other entities the co-processor already trusts (e.g. the manufacturer) vouch for the new entity. Thus, if a user wants to use the co-processor to run a program, the user must first establish trusts with several entities.

Nevertheless, this paper proposes an architecture that borrows several ideas from these systems. At the hardware level, Cerium relies on a 4758-like physically tamper-resistant CPU with a built-in private key. Unlike the 4758 co-processor, the Cerium CPU is the main processor in a computer and does not contain internal non-volatile storage. The Cerium CPU caches a portion of

a running program's instructions and data in its internal, trusted, cache. The remaining portions reside in untrusted external memory. Like Dyad [12], Cerium runs a μ-kernel in the secure CPU. The kernel's instructions and its crucial data are pinned inside the secure CPU's cache, so they cannot be tampered with. User-level processes that implement traditional OS abstractions (e.g. Mach servers) and virtualized operating systems (e.g. Windows running in VMWare) complete the μ-kernel-like operating system.

The Cerium CPU and the μ-kernel cooperate to protection programs from each other and hardware attacks. The μ-kernel partitions programs into separate address spaces, and the CPU applies conventional memory protection to prevent a program from issuing instructions that access or affect another program's data (cached or not). The CPU traps to the kernel when loading or evicting a cache line, and the kernel's trap handler cryptographically authenticates and copy-protects each program's instructions and data when they are stored in untrusted external memory. This technique allows the kernel to detect tampering of data stored off-chip.

The Cerium CPU reports what program is running and what hardware and software environment surrounds a program through certificates signed with the CPU's private key. The μ-kernel keeps a signature of the running program and includes the signature in the certificate. With a certificate, a user can detect if a different program binary was executed or if the computer is using a buggy kernel that cannot be trusted to protect the user's program.

Remaining sections of the paper describe system goals, related work, design, and applications.

2 System Goals

The main goal of the Cerium architecture is to provide tamper-evident execution of programs. Cerium protects a program's instructions, data, and control-flow during the program's execution so that they cannot be tampered by hardware attacks or other programs undetected. Another goal of Cerium is to allow trusted and untrusted operating system and processes to co-exist, all on the same CPU. Cerium reports the hardware and software environment to users so they can decide if the output of a program is in fact produced by that program. To help understand the design requirements, we describe a few examples.

Secure Remote Execution: Distributed execution of CPU-intensive programs can increase performance. Projects such as SETI@home [1] tap CPU cycles on idle computers scattered throughout the Internet. A problem with this computing model is that users cannot easily verify results obtained from an untrusted computer. A malicious user can, for example, return forged results without running the program. Cerium solves this problem by allowing a user to verify that an output is in fact produced by the user's program.

Copy Protection: Content distributors can use Cerium to enforce certain copyright restrictions. For example, an e-book's author can require customers to use Cerium, and distribute copies of the author's book so that each copy can only be viewed on a computer with a particular software configuration (i.e. with a given BIOS, boot loader, μ-kernel, and media player). A distributor can discover a Cerium computer's configuration from a certificate signed by the computer's secure CPU.

Secure Terminal: Users frequently check their e-mail by connecting to remote servers using untrusted terminals (e.g. at an Internet cafe). Although using tools such as `ssh` or SSL-based web login prevents passive adversaries sniffing data on a network, it does not prevent a bogus software on the untrusted terminal from stealing data. Cerium enables more secure login from untrusted terminals using the Cerium architecture. A user's trusted server authenticates the login software and the terminal's operating system, to make sure that the login software and the operating system appear on a list of software known not to steal data. As a result, the terminal approaches the safety of the user's own laptop.

3 Related Work

Previous research in secure processors and co-processors makes the use of a tamper-resistant CPU realistic. μABYSS [10], Citadel [11], and the IBM 4758 secure co-processor [7] place CPU, DRAM, battery-backed RAM, and FLASH ROM in a physically tamper-resistant package such that any tamper attempt causes secrets stored in the DRAM or battery-backed RAM to be erased. AEGIS [4] uses a processor that ties a secret to the statistical variations in the delays of gates and wires in the processor; an attack on the processor causes a chance in the processor's physical property, and therefore the secret. While no provably tamper-proof system exists, we believe current practices in building secure processors make physical attacks difficult and costly.

Dyad [12] and the IBM 4758 system [7] use tamper-resistant co-processors to provide trusted computing environments. Both systems allow only software from entities the co-processor trusts to run inside the co-

processor. The co-processor boots in stages, starting with the BIOS stored in the ROM. The software at each stage self checks its integrity against a signature stored in the co-processor's non-volatile memory. Each stage also authenticates the software for the next stage. Trusted entities install and maintain the software and their signatures. The co-processor establishes trust with an entity (e.g. a bank) if other entities the co-processor already trusts (e.g. the manufacturer) vouch for it. A trusted program running on the co-processor can also store encrypted data on external memory or disk. An advantage of the co-processor approach is that a user can connect a trusted co-processor using PCMCIA or USB; the user does not have to trust the microprocessor or the co-processor inside the computer the user is using. On the other hand, to write a program for a co-processor, the programmer must first establish trusts with several entities. Cerium provides a more open and flexible computing base. The Cerium architecture allows anyone to write operating systems and programs (buggy or not) for the secure CPU.

XOM [5] and AEGIS [8] also use physically tamper-resistant processors to support tamper and copy-evident computing. XOM and AEGIS do not to trust the operating system to protect programs from each other. Instead, the secure processor partitions cache entries and memory pages of different programs and the operating system in hardware/firmware. In contrast, Cerium depends on a μ-kernel to partition programs into separate address spaces and to authenticate and copy-protect each program's instructions and data when they are stored in untrusted external memory. Cerium securely identifies the μ-kernel, so a user can decide if the μ-kernel can be trusted to protect the user's program.

TCPA [2] promises to provide a trusted computing platform. TCPA uses a secure co-processor to store secrets and to perform cryptographic operations, but runs programs on a conventional microprocessor. Consequently, attacks on the DRAM and memory bus can alter the execution of a program undetected. Like Cerium, TCPA computes signatures of the system software as it boots up, and uses these signatures to enforce copy-protection. Palladium [3] is a software architecture that uses TCPA hardware. Palladium uses a small μ-kernel to manage applications that require security, much like Cerium.

4 Cerium

Cerium protects a program's execution using several techniques. Cerium relies on a physically tamper-resistant CPU with a built-in private key. The CPU runs all the software a computer uses. The CPU's tamper-resistant package protects a program's instructions and data from hardware attacks when they reside in the CPU's internal cache. A μ-kernel partitions programs into separate address spaces, and the CPU applies conventional memory protection to prevent a program from issuing instructions that affect data in another address space. The CPU traps to the kernel when loading or evicting a cache line, so the kernel can use cryptographic techniques to detect tampering of data stored off-chip. Upon request, Cerium tells a user what program is running, and what hardware and software environment surrounds the program, so the user can decide whether to trust the output of a program.

4.1 Tamper-Resistant CPU

The main component of Cerium is a tamper-resistant CPU, packaged in a way that physical attempts to read or modify information inside the CPU cannot succeed easily [4, 7, 10]. The CPU's tamper-resistant package protects its internal components, such as registers and cache, from hardware attacks. The CPU's internal cache is big enough (e.g. tens of megabytes) to contain a μ-kernel and the working set of data for most programs, but not big enough to contain whole programs. Programs that require more memory use untrusted external memory. The CPU traps to a kernel when evicting or loading a cache line, so the kernel trap handler can protect data stored in external memory (see Section 4.2).

Each CPU has a corresponding public-private key pair. The private key is hidden in the CPU and never revealed to anyone else (including software that runs on the CPU). The CPU reveals the public key in a *CPU certificate* signed by the manufacturer. A user trusts a CPU if the CPU certificate is signed by a trusted manufacturer. A CPU uses the private key to sign and decrypt data, and users use the public key to verify signatures and encrypt data for the CPU.

4.2 Operating System

At the software level, Cerium uses a μ-kernel to create and schedule processes onto the CPU and to handle traps and interrupts. The kernel runs in privileged mode; it can read or write all physical memory locations. Using page tables, the kernel partitions user-level processes into separate address spaces. The CPU applies conventional memory protection to prevent a program from issuing instructions that access or affect data in another address space. The page table of each address space is protected so only the kernel can change it. To prevent tampering of

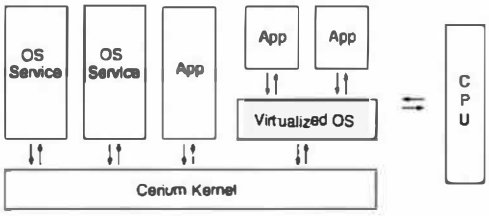


Figure 1: The organization of a Cerium system.

the kernel, the kernel text and some of its data reside in the secure CPU's cache and cannot be evicted.

The secure CPU traps to the μ-kernel when evicting data from its cache to external memory, or loading data from external memory into its cache. The kernel's trap handler decides, on a per-program basis, if and how the data should be protected. A program may ask the kernel to only authenticate or to both authenticate and copy-protect its instructions and data whenever they leave the secure CPU's internal cache. This technique is similar to the cryptographic paging technique used in Dyad [12].

The overhead of taking a trap on every cache event depends on a program's memory access pattern and miss rate, the cost of cryptographic operations, and the level of security a program demands. We believe that an increase in the size of a CPU's cache and using hardware-assisted cryptographic operations decrease the overhead. Furthermore, only programs that require integrity and/or privay protection take on these costs. We are currently investigating this issue.

Figure 1 shows what a Cerium system would look like. The μ-kernel and some user-level servers that implement OS abstractions form a complete operating system. Users can also run virtualized operating systems (e.g. Windows running in VMWare) in user space.

4.3 Booting

Cerium reports the hardware and software configuration of a computer, so a user can decide if the hardware and software can be trusted to protect the user's program. The CPU identifies each μ-kernel by the content hash of the kernel's text and initialized data segment. If the kernel is modified before the system boots, its content hash would change. Because a kernel's text and data reside inside the tamper-resistant CPU, they cannot be changed (e.g. by a DMA device) after the system boots. We refer to the kernel's content hash as the *kernel signature.*

A Cerium computer boots in several stages. On a hardware reset, the CPU computes the content hash of the BIOS and jumps to the BIOS code. Next, the BIOS computes the content hash of the boot loader, stored in the first sector of the computer's master hard drive, and jumps to the boot loader code. Finally, the boot loader code computes the μ-kernel signature, and jumps into the μ-kernel. Each stage uses a privileged CPU instruction to compute the content hash. The instruction stores the content hash in a register inside the CPU. The registers are protected so malicious programs cannot modify their content. This booting technique is similar to that of TCPA [2].

4.4 Running a Program

A program specifies its protection policy in its program header, so the μ-kernel knows how to protect the program's instructions and data when they are stored in external memory. There are three protection policies: no protection, authentication only, and copy-protection. A program asks the kernel to only authenticate its instructions and data if they do not need to be hidden from other programs. A program can also ask the kernel to copy-protect its instructions and data, so other programs cannot read their content.

When a program starts, the kernel first computes the content hash of the program's text and initialized data segment. The kernel uses this content hash as the *program signature.* A program signature uniquely identifies the program; if the program text or initialized data values change, the content hash would change as well. This technique only correctly protects programs using statically linked libraries.

4.5 Memory Authentication

This section describes how a μ-kernel handles authentication of data stored in untrusted DRAM. The kernel divides the entire physical address space into two regions. The *off-chip* region contains program data, such as text, data, and execution states, and some kernel data, such as page tables. The *on-chip region* contains the kernel's text, initialized data segment, and some data the kernel uses to authenticate data from the off-chip region. The on-chip region is pinned inside the secure CPU's cache so they cannot be evicted to external memory. Data from the off-chip region may be stored in external memory.

The μ-kernel efficiently authenticates data stored in external memory using a Merkle tree [6, 9]. A Merkle tree is a tree of hashes. Each intermediate node in the tree contains an array of *PA, hash* pairs, where PA specifies the physical address of one of the node's children, and hash is the cryptographic content hash of that child. Each leaf node stores hashes of data in external memory,

one hash for every 4K block (the size of a cache line on the CPU; this is acceptable because the CPU contains a large cache (e.g. tens of megabytes)). The root of the tree is stored in the on-chip region of the memory, so it cannot be modified by other programs or hardware attacks.

When the CPU traps to kernel to load data from external memory, the trap handler takes as its argument the physical address of the data. The trap handler uses the physical address to index the Merkle tree and find the corresponding leaf node. The trap handler computes the content hash of the data loaded into cache, making sure that the hash matches the one stored in the leaf node. When the leaf node is loaded into the cache, the trap handler verifies its integrity using the hash stored in the node's parent. This recursive process stops at the root of the tree. When the CPU evicts data from its cache, the kernel trap handler updates the Merkle tree accordingly.

4.6 Copy Protection

The CPU and μ-kernel can copy-protect a program's instructions and data while they are stored in external memory or on disk. We now describe how Cerium copy-protects a file, which could be a program's text or data.

Each copy-protected file has a corresponding *protection profile*. The protection profile contains a symmetric encryption key and signatures of trusted BIOS programs, boot loaders, kernels, and programs. A user encrypts the plaintext file using the symmetric key in the protection profile, then encrypts the profile using the Cerium CPU's public key. To open a copy-protected file, a program issues an instruction to ask the CPU to retrieve the symmetric key from the file's encrypted protection profile. The CPU returns the symmetric key only if the protection profile contains the current software configuration (i.e. signatures of the BIOS, boot loader, kernel and program). For example, the CPU refuses to return the decryption key if a malicious kernel, whose signature does not appear in the profile, is running. This technique is also used by TCPA and Palladium [2, 3].

A shell program that loads a copy-protected program stores the symmetric key of the new program in the on-chip memory region. When the CPU traps to kernel to load or evict a cache line for the new program, the trap handler uses this key to decrypt or encrypt the cache line.

4.7 Certifying Execution

Cerium is designed to be open and flexible. Cerium allows any μ-kernel or program to run on a computer, but reports what program is running and what hardware and software environment surrounds the program. A user can then decide if the identified hardware and software can be trusted to protect the user's program.

Cerium reports a computer hardware and software configuration in an *execution certificate*. Each execution certificate contains the CPU certificate, the content hashes of the BIOS and the boot loader code, the kernel signature, the program signature, and any data the program wants signed. For example, a program may also ask the kernel for content hashes of user-level OS services the program depends on. On a system call, the Cerium kernel creates a certificate and fills in the program signature and program data. The CPU fills in the rest of the certificate and signs the certificate with its private key.

Upon receiving a certificate, a user first extracts the CPU's public key and verifies that the CPU is made by a trusted manufacturer. The user also checks the certificate's signature. Next, the user checks if the signatures of the BIOS, boot loader, and μ-kernel in the certificate appear on a list of software the user trusts. Finally, the user checks if the program signature in the certificate matches the user's program. If the user trusts the CPU to correctly compute the hash of the BIOS, the BIOS to correctly load the boot loader and compute its hash, the boot loader to correctly load the μ-kernel and compute its signature, and the μ-kernel to correctly protect the user's program, then the user can trust the output of the program identified in the certificate.

5 Application Solutions

Secure Remote Execution: A user sends a program, the program's input, and a nonce to a remote computer. The program performs computation on the remote computer and obtains a signed execution certificate. The program includes in the certificate the hash of the nonce and the program's input and output. The program output and the certificate are then sent back to the user.

The fidelity of a program's output is determined in three steps. First, the user checks if the certificate identifies a trusted CPU manufacturer and a software configuration that the user trusts to protect the user's program. Second, the user checks if the certificate identifies a program signature that matches the signature of the user's program. Finally, the user checks if the hash of the nonce, the program input, and the received output matches the hash shown in the certificate. If all three conditions hold, then the output is in fact produced by the user's program.

Copy Protection: A content distributor takes three steps to copy-protect a file. First, the distributor sends a challenge to the customer's media player, and asks the media

player for an execution certificate that includes the hash of the challenge. The distributor uses the certificate to verify that the customer's hardware and software configurations can be trusted to not leave the copy-protected file unencrypted on disk or in external memory. Second, the distributor creates a protection profile and encrypts the file using the symmetric key in the profile. The distributor encrypts the profile using the public key of the customer's Cerium CPU. Finally, the distributor sends the encrypted file and the encrypted protection profile to the media player.

The media player asks the Cerium CPU to retrieve the decryption key from the protection profile. The Cerium CPU first decrypts the profile using its private key, then checks the current software configuration to make sure that it appears in the protection profile. If this check succeeds, the CPU returns the decryption key to the media player.

When the media player decrypts the encrypted file from the content distributor, the resulting plaintext initially resides in the secure CPU's cache. If the CPU evicts a block of the plaintext to external memory, the kernel's trap handler uses a session key that the media player generated to encrypt the evicted data.

Secure remote login: A user's trusted server must authenticate the hardware and software configuration of the untrusted terminal on behalf of the user. Before a login session, the login software (e.g. a `ssh` client) obtains a nonce from the trusted server and obtains an execution certificate from the untrusted terminal that includes the nonce. The login software forwards the certificate to the user's server. To verify the certificate, the server checks that the certificate is signed by a trusted manufacturer's CPU, that the untrusted terminal has a software configuration that can be trusted, and that the nonce in the certificate matches the one the server sent to the login software. If these conditions hold, the server returns a one-time response to the login software that the user can recognize as coming from the server. The user can then use the login software knowing that it will not steal any sensitive data. This solution does not guard against attacks on the computer's input interface, such as using a camera to monitor keyboard strokes.

6 Conclusion

This paper describes Cerium, a trusted computing architecture that provides tamper-evident program execution. Cerium uses a physically tamper-resistant CPU and a μ-kernel to protect programs from each other and from hardware attacks. Cerium reports what program is running and what hardware and software environment surrounds the program, so the a user can decide whether to trust a program's output.

Acknowledgments

We thank PDOS members and Satya for their comments.

References

[1] SETI@home. http://setiathome.ssl.berkeley.edu/.

[2] TCPA. http://www.trustedcomputing.org/.

[3] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft Palladium: A business overview, August 2002. Microsoft Press Release.

[4] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference*, December 2002.

[5] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168–177, 2000.

[6] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[7] S. W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[8] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. The AEGIS processor architecture for tamper-evident and tamper resistant processing. Technical Report LCS-TM-461, Massachusetts Institute of Technology, February 2003.

[9] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Hardware mechanisms for memory authentication. Technical Report LCS-TM-460, Massachusetts Institute of Technology, February 2003.

[10] S. Weingart. Physical security for the μABYSS system. In *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pages 38–51, 1987.

[11] S. White, S. Weingart, W. Arnold, and E. Palmer. Introduction to the Citadel architecture: security in physically exposed environments. Technical Report RC16672, IBM Thomas J. Watson Research Center, March 1991.

[12] B. Yee. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.

Hardware Works, Software Doesn't: Enforcing Modularity with Mondriaan Memory Protection

Emmett Witchel and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139

Abstract

Two big problems with operating systems written in unsafe languages are that they crash too often and that adding features becomes much more difficult over time. One cause of both of these problems is the lack of enforceable memory protection between module boundaries. Clear module boundaries make dependencies explicit, resulting in more reliable and maintainable code. Mondriaan Memory Protection (MMP) is a hardware/software design for fine-grained memory protection that can enforce module boundaries for systems written in unsafe languages. We present the design of an MMP-based modular operating system kernel and show how MMP can be used to provide module isolation while maintaining performance.

1 Introduction

Operating systems written in unsafe languages are efficient, but they crash too often. OS crashes are worse than user software crashes because an OS crash requires a time consuming reboot and may cause many users to lose data. The proliferation of embedded devices that manage non-transient data (like PDAs and digital cameras) translates lack of reliability into personal inconvenience. We believe system reliability should be a bigger goal for OS developers, and we believe that computer architects can do more to help OS developers write robust software.

The largest problem for OS reliability is device drivers, which according to one study, can have three to seven times as many bugs as the rest of the kernel [3]. Many operating systems, like Linux, load drivers into the address space of the running kernel. This makes calling them efficient because they share the kernel address space and run with full kernel privileges. But it also makes them dangerous, as a single driver bug can crash the whole system. Drivers are often buggy because writing a correct driver requires knowledge of poorly documented features of the kernel programming environment, and drivers are often written by device manufacturers who are not seasoned kernel developers.

Mondriaan Memory Protection (MMP) is a fine-grained hardware memory protection scheme that equips OS developers with a powerful and simple tool to increase reliability [7]. From an OS developer's perspective, MMP supersedes the protection part of a page table, providing permissions granularity down to single 32-bit words. It does not replace the page table structure, which is still needed if the system requires virtual address translation.

In this paper, we describe how MMP can be used to increase the robustness of an operating system, without compromising its performance. We can enforce the existing boundaries between dynamically loaded kernel modules and the core of the kernel, which is currently only protected by programmer convention. Although memory corruption is only one possible failure mode of a poorly behaved device driver (others include leaving interrupts disabled, breaking the lock discipline, or excessive resource consumption), it is the most common and the most difficult to guard against efficiently. Once boundaries with kernel modules are enforceable, we can begin dividing the core kernel into protected subcomponents to improve maintainability.

2 Mondriaan Memory Protection

MMP is an efficient, fine-grained memory protection system that allows word-level protection at word boundaries. The MMP architecture was described in detail in [7], but here we give a brief review.

Figure 1 shows the architectural structure of an MMP system. A *protection domain* (PD) contains a map from address to permissions for the entire address space. The map is stored in a *permissions table* held in kernel memory. The table is similar to the permissions part of a page table, but permissions are kept for individual words (32-bits) in an MMP system. A CPU control register holds the base address of the active domain's permissions table. MMP can be used in systems with a single virtual or physical address space, or systems with multiple virtual address spaces in which case a PD lives within one address space.

The CPU contains a hardware control register which

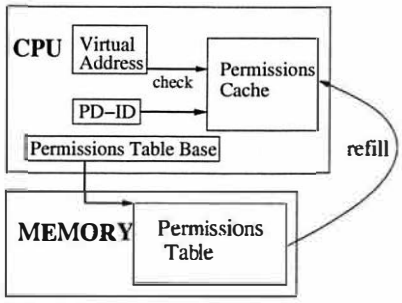


Figure 1: The major components of the Mondriaan memory protection system. On a memory reference or instruction execution, the processor checks a hardware permissions cache. If the permissions information is not in the cache, it is fetched from the table in memory.

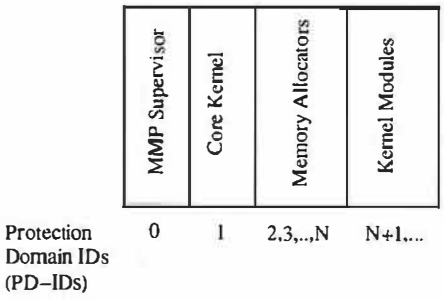


Figure 2: How the kernel address space can be divided into protection domains. The memory supervisor is in protection domain 0, and has unfettered access to memory. The bulk of the kernel is loaded into the first created protection domain (PD-ID 1). Then it loads other pieces of itself, like the memory allocators, into other domains. Finally kernel modules are loaded on demand.

holds the protection domain ID (PD-ID) within which the currently running thread executes [5]. The MMP hardware on the CPU checks every memory access to see if the active domain has appropriate permissions for that access. Every instruction fetch is also checked for execute permissions.

The processor maintains a hardware permissions cache, the *protection lookaside buffer* (PLB) [5], to accelerate permission checks. Detailed simulations show this cache to be effective, resulting in only an 8% increase in memory traffic due to PLB misses even when fine-grained permissions are used extensively [7]. Permissions checks only need to be completed before instruction commit and so are not on the critical path of the processor. We expect a modern out-of-order processor would hide some or all of the latency of the references to the permissions table, reducing the performance impact of the additional memory references.

Frequent permissions changes might require hardware support for PLB consistency in multiprocessors. The form of this support is still an open question.

MMP is backwards compatible with current instruction sets and binaries. CPUs could be made with TLB permissions and MMP permissions to allow full binary compatibility. Or MMP could replace TLB permissions with support from the device dependent layer of the kernel, and a few trusted applications like the system loader and the dynamic linker.

We preserve the user/kernel mode distinction, where kernel mode enables access to privileged control registers and privileged instructions. Access to privileged memory areas (like I/O space) can be controlled with MMP. The CPU encodes whether a domain is user or kernel mode using the high bit of the PD-ID control register (a zero high bit implies a kernel domain). Protection domain 0 is used to manage the permissions tables for other domains and is special in that it can access all of memory without the mediation of a permissions table.

3 Using MMP in the Kernel

In this section, we describe the design of an MMP-based operating system kernel.

3.1 Memory Supervisor

Figure 2 shows how the kernel address space can be split into multiple protection domains. The MMP memory supervisor domain (PD-ID 0) manages the memory permissions tables and provides an API to control memory permissions. In this section, we describe the MMP memory supervisor API by showing how it would be used during the boot of a modularized kernel.

At system reset, the processor starts running at the reset vector in PD-ID 0. The BIOS loads the memory supervisor into physical memory and transfers control to it, letting it know how much physical memory is in the machine. Early on, the supervisor establishes a handler for hardware permission faults.

The MMP hardware checks all processor memory accesses against the values stored in the current permissions table (except those made by domain 0). The MMP supervisor software can enforce additional memory usage policies because all calls for permissions manipulation are made via the supervisor. The supervisor will reject requests that violate its policy. Just because the supervisor exports an API does not mean that all created domains have permissions to call into it. As we will see, it is possible to construct a domain which does not have permission to call into the supervisor, forcing memory management to happen via the intermediary of the creating domain.

For example, the supervisor tracks *ownership* of memory regions. A domain obtains ownership after allocating a new memory region from the supervisor,

or when another domain grants ownership of a region. Only the owner of a memory region may revoke permissions, or grant ownership to another domain.

Once initialized, the supervisor creates a new domain (PD-ID=1) to hold code and data for the core of the kernel. Protection domain creation is provided by the `mmp_alloc_PD(user/kernel)` supervisor call, which returns a PD-ID. The supervisor does not allow a user domain to create a kernel domain.

To start the kernel, the supervisor must first load the boot loader into PD 1. Initially, a PD has no permissions to access memory. In order for the boot loader to run, it will need execute permission on its code, read and write permissions on its data, a read-write stack, and possibly a read-write heap. Setting permissions is done by the `mmp_set_perm(ptr, length, perm, PD-ID)` routine. The memory supervisor uses the `mmp_set_perm` call to establish proper permissions for boot loader execution. The supervisor then performs a cross-domain call (described below) to transfer control to the boot loader which now runs in a protected kernel domain (PD-ID=1).

The boot loader wants to load the core kernel, and so needs to ask the memory supervisor for additional memory space. The `mmp_alloc(n.bytes)` supervisor function allocates a region of memory and returns a pointer (a variant of `mmp_alloc` allows a desired address placement to be supplied). The supervisor records PD-ID 1 as the owner of this memory region. The owner of a region can call `mmp_free(ptr)` to release a memory region back to the supervisor (`mmp_free` can also take an optional length parameter, allowing partial resources to be reclaimed).

Once the core kernel starts running, it can create child PDs to hold kernel modules. The core kernel will want to export permissions for portions of its address space to its child modules using the `mmp_set_perm` call, and it might also want to pass on ownership of memory regions to kernel modules, to allow them to manage the permissions of their children. The `mmp_mem_chown(ptr, length, PD-ID)` call passes ownership of a memory region to the protection domain given by the PD-ID parameter. Although a kernel module could be allowed to ask the supervisor for memory regions directly, usually the core kernel will manage memory usage of its modules. The core kernel can block kernel modules from calling the memory supervisor by not exporting call permission on the supervisor entry points to the kernel modules.

The `mmp_set_perm` call supports a `transitive` flag which, in addition to exporting permissions, also allows the receiving domain to transitively export permissions. This allows calling domains to either enforce a policy of only allowing a particular service domain (perhaps one containing cryptographically verified code) to implement a function, or allowing a service domain to subcontract work to other service domains. Transitive permissions are still distinct from ownership because only the owner can return memory to domain 0, and a domain that receives transitive permissions can not revoke permissions from a domain higher on the receiving chain.

Protection domains are created hierarchically, and they are destroyed hierarchically. The supervisor tracks the entire protection domain hierarchy, allowing parents to call `mmp_free_PD(recursive)` on their children. If the recursive flag is true, all of the deleted protection domain's children are also deleted. Otherwise, they are reparented to the closest surviving parent remaining in the tree.

One important special case for sharing data is global read-only access. MMP supports export of data to all protection domains read-only. When a piece of memory is exported globally, the supervisor adds the permissions to all existing permissions tables. It also tracks the global export in a table so it can add permissions for this globally exported memory to new protection domains as they are created.

3.2 Stacks and threads

Code and heap data regions can be associated with a protection domain, and are typically owned by one domain and exported to others. Stack storage must be managed differently, however, because stacks are used by threads that move between protection domains. In our MMP OS design, stack storage is owned by the memory supervisor. To get stack storage, a thread manager in a protection domain calls `mmp_alloc(stack)`. The `stack` flag tells the supervisor that this is a stack segment, a fact which the supervisor records while maintaining ownership of the storage.

The memory supervisor only owns and manages the stack space for each thread. Other details about the thread, like its control block and the scheduling policy that govern it, are determined by the kernel or an arbitrary thread-managing domain.

Stack permissions are managed by the supervisor call `mmp_supr_set_perm(ptr, length, perm, exclusive, PD-ID)`, which is like the `mmp_set_perm` call, but the `mmp_set_perm` call only works for memory that is owned by the caller. The `mmp_supr_set_perm` call requests that the supervisor make a permissions change on memory that it owns. Of course, the supervisor range checks the address and refuses action if the request is inappropriate. The `exclusive` flag requests that all permissions for other PDs be revoked.

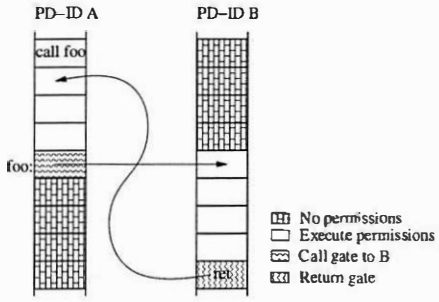


Figure 3: How MMP is used for cross-domain calling. The arrows indicate a domain crossing, which is handled by hardware. The call gate has the destination PD-ID stored in a special record in the MMP protection table. The return gate verifies that it is returning to the last call on the cross-domain call stack (not pictured).

When a stack segment is allocated, the supervisor records the creating protection domain, and a stack-ID, which is just the base address of the stack segment. When a thread is scheduled on a CPU, the thread manager must make the supervisor call `mmp_set_stack(stack_seg, cpuid)` to tell it that a certain stack is now active on a certain CPU. The supervisor checks that this thread manager has permissions to make this stack segment active. When the supervisor receives a call to set stack permissions, it checks that the request is for the active stack.

Although this scheme prevents one thread changing permissions on another thread's stack, when multiple threads run in the same protection domain they can still potentially access each other's stack frames. This is a minor protection violation, as the code in the domain has already been trusted with the stack frames. But we can eliminate this violation by adding another hardware mechanism, discussed in Section 3.3.

3.3 Cross-domain calls

Threads move between protection domains by performing a cross-domain call as shown in Figure 3. The cross domain call can be initiated by any control flow instruction, though it will usually be a standard subroutine call instruction. The target of the call is marked with a special permissions value known as a *call gate*. A call gate also has the PD-ID of the target protection domain stored in a special record type in the permissions table.

When call gate permissions are detected on a subroutine call, the hardware atomically pushes the return address and the caller's PD-ID onto a stack that resides in protected storage. This stack is called the *cross-domain call stack* and is implemented with some combination of an on-chip top-of-stack buffer, backed up by off-chip protected memory. The architecture then reads the new (callee's) PD-ID value from the permissions table and copies this into the CPU's PD-ID register. It looks up the protection table base pointer for the new PD-ID, and stores it in the table base register. At the end of this process, instructions are fetched from the context of the new protection domain.

MMP also uses return gates, which are the duals of call gates. They are also implemented using standard instructions and special MMP protection values. A return gate causes the architecture to pop the cross-domain call stack, finding the return address and protection domain of the last call. The architecture checks the return address against the return address being used by the return instruction. If they are different, a fault is generated which is handled by the memory supervisor. If the return addresses match, the hardware sets the protection domain to the PD-ID that was popped off the stack.

Call and return gates provide an efficient mechanism for mutually distrustful protection domains to safely call each other's services, without requiring new instructions in the ISA. Cross-domain calls are analogous to lightweight remote procedure calls, though cross-domain calls do not require copying data for protection, or an argument stack per domain pair, as LRPC does.

We expect cross-domain calls to be fast because the amount of on-chip state that needs to be changed is small. We believe CPU designers will be motivated to accelerate cross-domain calls to enable the benefits of protected execution. For example, traditional CPU microarchitectures flush pipelines on a context switch, imposing a large overhead. Domain switches can be made considerably faster by associating PD-ID values with each instruction in the pipeline, reducing the need to flush the pipeline.

If the called function needs an activation frame, it must request permissions for the stack space, and also make sure that permissions for the frame are exclusive to the current thread. This is done using the `exclusive` flag in a call to `mmp_supr_set_perm`. Because domains take exclusive access to a frame before executing in the frame, a frame's permissions do not need to be revoked at the end of a function for the caller's safety. A callee that is concerned about security could overwrite its activation frame before returning to avoid leaking information.

Calls to establish a frame will be frequent and could potentially be expensive. Two special hardware registers can make the creation of a frame fast, and can make permissions to read and write the frame thread-local, closing the security loophole discussed in Section 3.2.

When the supervisor makes a stack current for a given CPU, it fills in two registers—frame base **fb**, and stack limit **sl**. The hardware allows read and writes to addresses between **sl** and **fb** (stacks grow down so **sl** $\leq$

fb). The **fb** value points to the base of the current activation frame. Its initial value for a given thread's quantum is specified (as a parameter to `mmp_set_stack`) when the thread manager starts the thread. The memory supervisor verifies the initial value of **fb** to make sure it is within the stack segment that is being activated. On a cross-domain call, the current **fb** is pushed onto the cross-domain call stack, and the current stack pointer is made the new **fb**. The hardware checks that the new **fb** value is smaller than the old value. Thus the hardware insures that the stack grows down, and the memory supervisor insures that it starts and ends in the right place, so the two registers can only be used to gain permission to read and write stack memory. The registers become part of the thread state which must be saved and restored.

3.3.1 Passing arguments

Heap data is owned by a protection domain, so cross-domain sharing of heap data is straightforward—the caller exports permissions to the callee. This is a lightweight form of argument marshaling that does not require data copying or even data structure traversal (for many data structures). Domains can set up shared buffers in advance of the cross-domain call. In a producer-consumer relationship, the producer would maintain read-write access on a buffer and flag value, while the consumer has read-only access on the buffer and read-write access on the flag. Once the permissions are established, they do not need to be modified for every call.

Passing arguments on the stack is more complicated because a protection domain does not own the stack. To pass arguments on the stack, a cross-domain call must be preceded by a call to the supervisor to export any permissions that might be required for the call to work properly, e.g., giving read-write permissions on a stack structure whose address is being passed as a parameter.

Calls to export permissions on a stack frame are stylized and so can be highly optimized. For instance many custom entry points could be provided which take the current top of the stack as the only parameter. They would establish read-only permission for a fixed small number of stack slots by writing directly into the permissions tables. Most calls would fit into one of these patterns, but for those that did not, the dynamic linker could request the generation of a custom entry point for a given stack layout.

We can reduce the number of exports for inter-module calls by hoisting the export out of a loop, reusing stack slots for different inter-module calls, and not changing the permissions until right before the call if the call is unlikely to execute.

3.4 Space overhead of protection domains

For its finest-grained permissions tables, MMP stores two bits of permissions data per 32-bit word, so the space cost for the tables is $\frac{1}{16}$, or 6.25%. Applications that use coarse-grained permissions regions can experience less than 6.25% space overhead, potentially much less (e.g less than 0.7% for putting each program section in its own protection region [7]). The space overhead of the tables is proportional to how densely the address space is being used, with lower density leading to higher overhead (just as with page tables).

If multiple protection domains are arranged densely in the address space (as kernel module code and data are arranged in Linux), then there is little additional space cost to dividing domains. Each new domain requires a root table, which has a fixed cost of 4KB (though they can be made smaller if need be). The root tables need to be stored in unmapped kernel memory, but user root tables can be swapped. Domains that share a permissions view for much of memory can share permissions tables below the root level.

4 Adding MMP to Linux

We split the Debian Linux kernel version 2.4.19 into different domains, putting the core kernel in one domain and placing each loaded kernel module in its own, separate, domain. Code and data exports were derived from tools that interpreted the symbol information in the kernel modules. Because so little code and data is actually imported or exported by any module (relative to what is available in the kernel address space), restricting access to those symbols results in a large gain in modularity.

For instance, most modules import the kernel function `printk` so they can log errors. We treat the unresolved symbol in the module as a request for permissions to call the routine. While this works well for code symbols, data boundaries are less likely to be completely characterized by symbol information because a module might dereference a pointer from an imported structure, reading memory outside that structure.

We booted the OS on bochs, a complete machine simulator, and measured domain crossings. Our rough prototype implemented all of MMP in the hardware model (including table management, which really belongs in OS code). The OS boot from disk shares many properties with any disk intensive workload. There were 284,822 protection domain changes in the boot, 97.5% of which were to or from the IDE driver. About 1 billion instructions were executed (955,240,000), yielding an average of 3,353 instructions executed in each domain. This demonstrates a surprisingly fine-grained interleaving of module execution and underscores the need for

efficient cross-domain calling, justifying hardware support.

MMP not only enforces memory safety, it enables performance optimizations. For instance, one reason the kernel needs a copyin procedure is because it can not trust the user to put their data in the right spot and not corrupt kernel data structures. With MMP, we can change the interface to allow the user to write their data into kernel space directly, and still protect kernel data structures.

5 Comparison with other protection mechanisms

Nooks [6] provides device driver safety using conventional hardware. MMP can reduce the sometimes substantial performance overheads Nooks endures to run on current hardware. Also, MMP has many uses in addition to device driver safety.

Palladium [2] used x86 hardware protection for intermodule protection both in the kernel and at user level. It is very difficult to use x86 segmentation to implement protected shared libraries (Palladium used page-based protection for shared libraries), and using it in the kernel complicates the programming model for extensions.

Capabilities [4] are a fine-grained protection mechanism that OS designers have used to build big systems (e.g., IBM's AS400). Capabilities are special pointers that contains both location and protection information for a segment. Capabilities have known disadvantages such as difficulty with rights revocation, requiring tagged memory, and difficulty for two domains to share a data structure (with embedded capabilities) with different permissions.

Lightweight remote procedure call (LRPC) [1] enables modular boundaries for unsafe languages, using a software-enforced discipline for protected calling. It allows the partitioning of an OS into different protection domains whose interactions are protected, but LRPC achieves this protection by using data marshaling and copying, a costly process which MMP avoids. Data copying is inefficient, and imposes a minimum size on a protection domain so calls to the domain can be amortized.

There are a variety of safe language approaches for OS extensibility and all of these approaches have common problems—excessive CPU and memory consumption is common in safe languages or unsafe languages retrofited with type information. A safe language restricts an implementation to a single language, it ignores a large base of existent code, the analysis needed to establish type-safety can be global and thus difficult to scale, and type-safe languages often need unsafe extensions to manage devices.

A deeper problem with language-only safety is the size of the system that must be trusted. For an MMP system, one must trust the MMP hardware and the MMP supervisor software. These are likely to be much simpler and more amenable to verification than a language compiler and runtime. This is especially true for optimized safe-language implementations which employ complex analyses to improve runtime efficiency.

Modern static analysis and model checking tools can scale sufficiently to deal with large OS codes. These systems can find many important bugs without flooding the user with false positives. But they do suffer from false negatives, and are therefore compatible with and benefit from the dynamic checking of an MMP system.

6 Conclusion

MMP allows systems to be extensible, efficient, and robust, whereas current software-based schemes require that a designer choose only two of these properties. Compared with other proposed hardware fine-grained protection schemes, MMP has the advantage that it is backwards compatible with existing instruction sets and existing OS protection models, and so can be introduced incrementally.

MMP enables the hardware to enforce the intermodule boundaries already present in the software structure, helping to address the problem of poor OS stability due to poorly coded device drivers. MMP supports further modularization of the kernel by reducing the overhead of enforced modularity, which should result in systems that are more reliable and more maintainable.

References

[1] E.D. Lazowska B.N. Bershad, T.E. Anderson and H.M. Levy. Lightweight remote procedure call. In *SOSP-12*, 1989.

[2] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *SOSP-17*, 1999.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems error. In *SOSP-18*, 2001.

[4] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9(3):143–155, March 1966.

[5] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *ASPLOS-V*, 1992.

[6] M. Swift, S. Martin, H. M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings SIGOPS-10*, 2002.

[7] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS-X*, Oct 2002.

Flexible OS Support and Applications for Trusted Computing

Tal Garfinkel Mendel Rosenblum Dan Boneh
{talg,mendel,dabo}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

Trusted computing (e.g. TCPA and Microsoft's Next-Generation Secure Computing Base) has been one of the most talked about and least understood technologies in the computing community over the past year. The capabilities trusted computing provides have the potential to radically improve the security and robustness of distributed systems. Unfortunately, the debate over its application to digital rights management has caused its significant other applications to be largely overlooked. In this paper we present a broader vision for trusted computing. We give an intuitive model for understanding the capabilities and limitations of the mechanisms provided by trusted computing. We describe a flexible OS architecture to support trusted computing. We present a range of practical applications that illustrate how trusted computing can be used to improve security and robustness in distributed systems.

1 Introduction

Many difficult problems in today's distributed systems, such as preventing denial of service, performing access control and monitoring, and achieving scalability, are either caused or severely exacerbated by the fact that clients are untrusted and thus potentially malicious. This forces system designers to implement most system policy and sensitive computations in the core of the system, where trust resides, instead of at the endpoints where most of the system's resources and capabilities are. The only complete solution to this problem has been the use of closed platforms, such as those in cellular networks and banking systems, where special-purpose, tamper-resistant clients are utilized that provide end-to-end trust. This approach has demonstrated significant benefits, allowing the construction of some of today's most capable and robust distributed systems. Unfortunately, this approach presently necessitates the use of dedicated hardware, thus limiting designers to the use of only a few types of devices over which they must have exclusive control.

In the near future it will no longer be necessary to force designers to make trade-offs between the benefits of open and closed platforms. This change will come as the result of ubiquitous support for trusted computing platforms. Trusted platforms will allow systems to extend trust to clients running on these platforms, thus providing the benefits of open platforms: wide availability, diverse hardware types, and the ability to run many applications from many mutually distrusting sources while still retaining trust in clients.

The vision of trusted platforms cannot be achieved with today's operating systems which offer poor assurance and implement a security model that is largely orthogonal to that required for trusted computing. To meet the demands of implementing a trusted platform we outline the design of a new OS architecture based on the idea of a trusted virtual machine monitor. In this model, traditional applications and OSes can run side-by-side on the same platform in either an "open box" or "closed box" execution model in keeping with the trust requirements imposed by the application.

In the next section we define and describe the components that make up trusted computing. In Section 3 we present our approach of using a trusted virtual machine monitor to support a mixture of open and closed box models simultaneously. In Section 4 we examine a selection of practical areas where trusted computing can provide novel functionality yielding significant benefits for security, scalability and robustness. Section 5 discusses related work.

2 Trusted Platforms

Open platforms are general-purpose computing platforms where there is no apriori trust established between the hardware of the platform and a third party, that could be used to prove the functionality of the platform. Examples of these include workstations, mainframes, PDAs, and PCs. Open platforms possess many practical benefits over closed platforms. Unfortunately a remote party cannot make any assumptions about how that platform

will behave or misbehave.

Closed platforms are special-purpose computing devices that interact with the user via a restricted interface (e.g. automated tellers, game consoles, and satellite receivers). A closed platform can authenticate itself as an authorized platform to a remote party using a secret key embedded in the platform during manufacturing. Closed platforms rely on hardware tamper resistance to protect the embedded secret key and ensure well-behaved operation.

Trusted platforms provide the best properties of open and closed platforms. As with an open platform, trusted platforms allow applications from many different sources to run on the same platform. As with a closed platform, remote parties can determine what software is running on a platform and thus determine whether to expect the platform to be well behaved. The process of dynamically establishing that a platform conforms to the specification expected by a remote party is done through a process called attestation.

Attestation consists of several steps of cryptographic authentication by which the specification for each layer of the platform is checked from the hardware up to the operating system and application code. At a high level, the steps in a basic model of attestation are as follows. A more detailed example is given in Section 4:

- A hardware platform has a signing key K_{sign}. It also has a public key certificate (C_{hw}) for this key.
- When an application A is started it first generates a public/private key pair PK_A/SK_A. Next, the application requests the platform to certify its public key PK_A. The platform uses its signing key K_{sign} to generate a certificate for PK_A. We denote this certificate by C_A. Along with standard certificate fields, the certificate C_A contains the *hash of the executable image of the application A*. This hash is at the heart of the attestation process. The signed certificate C_A is returned to the application.
- When the application A wants to attest its validity to a remote server it sends the certificate chain (C_{hw}, C_A) to the remote server. The server checks two things:
  - The signatures on both certificates are valid and C_{hw} is not revoked, and
  - The application hash embedded in C_A is on the server's list of applications it trusts.

  At this point the server is assured that C_A comes from an application it trusts. The application can now authenticate itself by proving knowledge of SK_A. For example, the application and the remote server could run an authenticated key exchange to generate a shared session key. All communication between the remote server and the application will be protected using this key.

We emphasize that *attestation must result in a shared secret between the application and remote party*, otherwise the platform is vulnerable to session hijacking—an attacker could wait for attestation to complete, reboot the machine into untrusted mode, and masquerade as an authorized application.

Leveraging attestation requires the presence of software that allows the remote party to meaningfully interpret the state of the system. This takes place through a multi-step process whereby the hardware will attest to what operating system it booted, the operating system will in turn attest what application it requires a key for, and will only allow the use of that key by that given application.

Limitations of attestation. It is important to realize that software attestation only tells a remote party exactly what executable code was launched on a platform and establishes a session key for future interaction with that software component on the platform. This does not provide trustworthiness in the usual sense:

- The software component could be buggy and produce incorrect results. The onus is on the remote party to choose who to trust.
- Attestation provides no information about the current state of the running system. For example, attestation does not show whether the software component has been compromised by a buffer overflow attack, infected by a virus, etc.
- Future behavior can only be ensured for authenticated interactions via a shared secret.
- A platform is only as trusted as the tamper resistance of hardware and level of assurance of its trusted OS.

3 An OS for Trusted Platforms

The vision of trusted computing falls apart when it encounters the realities of modern general-purpose operating systems. OSes such as Microsoft Windows and Linux are large and complex code bases optimized over the years for ease of use, performance, and reliability. As a result they are incompatible both in design and implementation with the objective of providing a high assurance platform. High assurance is essential as a trusted OS must instill confidence in remote parties that it can be relied upon to execute their code in a well-specified fashion.

The protection model provided by contemporary operating systems is poorly matched to the needs of trusted computing. In a trusted platform the primary security objective is to isolate subjects from one another. The fine-grained resource abstractions for controlled sharing provided by typical OSes would add needless complexity to a trusted OS, thus detracting from its primary goal of providing secure isolation.

The approach we advocate and have begun to explore in our own work on building a trusted operating system, is to use a virtual machine monitor [14] (VMM). A virtual machine monitor is a thin system software layer that exports the abstractions of virtual machines (VMs) that look like the real hardware.

The simplicity of the VMMs interface and implementation provides the means for building a high-assurance OS that offers strong isolation [17]. VMM's also provide backwards compatibility, allowing existing services and operating systems to realize the benefits provided by trusted platforms with little or no modification. Users can continue to use their normal operating systems for applications that do not require trust from a remote party. Developers building services that require trust can utilize the wide range of existing secure operating systems, applications, etc. , thus allowing them to leverage a huge amount of high quality existing code and development environments.

Our trusted virtual machine monitor (T-VMM) exports two different types of virtual machine abstractions:

Open-box VMs are traditional virtual machines that exactly match the hardware interface of the machine. They are used to run general-purpose operating systems such as Microsoft Windows or Linux and allow the platform owner full access to the hardware state of the VM just as in a normal open platform.

Closed-box VMs provide the same hardware interface as open-box VMs. In addition, a virtual device is provided that allows them to do attestation. To platform owners, the closed-box VMM is a black box. They can grant it access to resources but they cannot inspect or tamper with its contents.

Hardware attestation needs only attest to the fact that the T-VMM is running. For applications to attest, the attestation virtual device can provide a closed-box VM with a signed hash of its executable plus some attributes which it can then present to a remote party to obtain a token encrypted under the public key of the T-VMM. The attestation interface can then be used to decrypt this token, but it will only release the token to the VM whose hash and attributes match those that were originally used to request the token. This token will contain a session key, certificate, or some other means of allowing the VM to authenticate itself.

The T-VMM has total control of both the visibility and use of hardware resources by the VMs. Resource management policy is specified by the platform owner directly to the T-VMM.

Storage devices are abstracted into disjoint virtual disks. Virtual storage can be either encrypted at the block level by the T-VMM or left as plain text in accordance with the performance and security requirements of the VM. Communication devices such as network interface cards can either be virtualized or exported directly to a VM. User interface and display devices are multiplexed among the VMs in such a fashion that one VM cannot observe the user interactions of another.

To support composition of VMs and communicate between VMs, the T-VMM supports the notion of a virtual device. A virtual device can be implemented by a closed box VM and exported as a device to any VM. For example, many closed box VMs will want to export a virtual NIC or virtual serial port to allow other local VMs access to their functionality.

The T-VMM supports a trusted console that allows access to the T-VMM. This is used to control the allocating hardware sources to VMs, mapping of I/O devices to VMs, the destruction of VMs, etc. . The console VM can be accessed via a trusted path. How to securely facilitate this access in a backwards compatible and seamless way is a question we are still are working to address.

4 Example Applications

We survey several areas where trusted platforms promise to have significant impact. We discuss how the introduction of trusted platforms can significantly increase the functionality of existing client side technologies, such as distributed firewalls and massively distributed parallel computing clients. We also look at some entirely novel applications of this technology, like those facilitated by rate limiting. We do not discuss any applications related to Digital Rights Management (DRM) since we find them far less exciting that the applications discussed below.

Regulated Endpoints and Distributed Firewalls. Traditionally firewalls assume that everyone on the "inside" of the network is trusted, while everyone on the outside is untrusted. However, the increased use of wireless access points, tunnels, VPNs, and dial-ins breaks down the distinction between inside and outside. Given today's increasingly dynamic network topologies, distributed firewalls [7] greatly simplify the task of implementing network security policies. With a distributed firewall secu-

rity policy is defined centrally, but enforced at each individual network endpoint. This supports a richer set of policies and greater scalability than traditional centralized firewalls [15].

On standard hosts, distributed firewalls do an excellent job of protecting a host from others, but are of little use for protecting others from the host—there is no way of ensuring that the host does not simply tamper with or bypass the firewall.

On a trusted platform a distributed firewall is a significantly more powerful primitive since it can prevent packets that violate the central security policy from ever reaching the network in the first place. For example, the distributed firewall can prevent applications that attempt port scanning and IP spoofing from ever reaching the network. Similarly, the firewall can ensure that all VMs on the machine are properly implementing connection rate limits. Hence, distributed firewalls on trusted platforms can provide well-regulated endpoints for a wide variety of different network types.

The architecture for a distributed firewall on a trusted platform is as follows. The distributed firewall runs in its own closed-box VM and listens on a virtual NIC. All packets generated by open-box application VM's on the machine are sent to the distributed firewall VM. The distributed firewall ensures that these packets adhere to the security policy being enforced. If so, it embeds them into an IPsec packet and sends them to their destination on the network. If not, the packets are blocked. The termination point of the IPsec tunnel is the closest network gateway, or alternatively, the remote destination host. The IPsec tunnel is only used to prove to the IPsec endpoint device that the packets are sent via the firewall VM. Consequently, it suffices to use the Authentication Header (AH) in IPsec. There is no need to encrypt the packets.

The main question is how does the IPsec endpoint device know that the sending host is running a distributed firewall. At a high level, the idea is as follows: during initial firewall setup the distributed firewall VM uses attestation to convince a certification authority (CA) that it is an authorized firewall implementing the required security policy. The CA issues a certificate to the firewall VM enabling it to establish IPsec tunnels with peer devices. Without this certificate, peer devices will reject connection requests. Consequently, no application on the machine can communicate with a networked device unless it sends its packets through the firewall VM.

In reality, the exact firewall VM architecture is more complicated. We briefly explain the initial attestation protocol with the CA. We are assuming that the T-VMM on the machine has certified public/private key pairs that can be used for encryption/decryption and for signing. The following steps take place during initial firewall setup:

- the firewall VM generates a public/private key pair PK_{FW}/SK_{FW}.
- The firewall VM requests the T-VMM to sign the hash of the executable image running inside the firewall VM. Let S be the resulting signature. This signature is the main capability used for attestation.
- The firewall VM contacts a CA and sends the public key PK_{FW}, the signature S, and a certified T-VMM public key PK_{VMM}.
- The CA verifies that the firewall executable image (whose signature is S) is an authorized firewall. If so, it issues a certificate $CERT_{FW}$ for the firewall's public key PK_{FW}. The CA also embeds the hash of the firewall executable in the certificate. The CA encrypts the resulting certificate $CERT_{FW}$ under PK_{VMM} and sends the resulting ciphertext $E[CERT_{FW}]$ to the firewall.

This completes the initial firewall setup. Note that no open-box VM can directly use $E[CERT_{FW}]$ since it is encrypted using the T-VMM's public key. Whenever the firewall VM is launched, it first requests the T-VMM's virtual attestation device to decrypt $E[CERT_{FW}]$. The T-VMM does so only if the hash of the executable running in the VM matches the hash inside $CERT_{FW}$. If there is a match, the firewall VM obtains $CERT_{FW}$ which enables it to setup IPsec tunnels with remote hosts. Consequently, when a remote host receives an IPsec session request using $CERT_{FW}$ it is assured that the requesting machine is running an authorized firewall VM.

Rate Limiting for DDOS Prevention. Rate limiting can be used to address the problem of Distributed Denial of Service (DDOS) attacks at both the network and application levels. For example, by limiting the rate at which client machines can issue queries in a P2P network we defend against certain P2P DoS attacks [9]. By limiting the rate at which a machine can open network connection we defend against certain network DDOS attacks [16, 10]. Finally, by limiting the rate at which machines can send email we reduce the rate at which spam email is generated [11].

Implementing a rate limiter with a trusted platform is straightforward. On each trusted platform we run a ticket-granting service in a closed-box VM. The ticket-granting service issues at most one ticket every time quantum. These tickets are content dependent. For example, to limit the rate at which a P2P client issues queries we require an open-box P2P client VM to obtain a ticket from a ticket-granting VM for every query being sent. More precisely, prior to issuing a query, the P2P client VM sends a hash of the query to the ticket-

granting VM (via a virtual NIC). The resulting ticket is attached to the outgoing query. The P2P network will discard any incoming queries that contain no ticket or an invalid ticket. Consequently, each client machine can generate at most one query every time quantum (say every 5 seconds).

Without attestation the best known method for achieving these types of rate limits is using client puzzles [11, 4], the practice of forcing a client to perform some costly computation (solving a puzzle) for each request made. A trusted computing solution has several major advantages over client puzzles: no resources must be wasted in order to generate tickets (a real consideration on mobile devices where computing expensive client puzzles could present a significant power drain); users do not need to wait for tickets to be issued; client puzzles vary heavily in their impact based on the type of platform (processor and memory speeds, etc.) whereas trusted-computing based rate limiters are independent of device size or Moore's law.

Improving Robustness via Reputation. Understanding DDOS attacks on today's P2P storage systems requires considering a broad spectrum of attack types. One of the most insidious types of attacks are those based on content poisoning, where a user disseminates damaged or incomplete content (e.g. audio files which have artifacts inserted) in order to make the good content difficult or impossible to find amongst the noise.

One approach to solving these and other problems of mis-behaving users are the use of reputation systems. These are already widely seen in use in online games, P2P file sharing systems [2], and even on eBay to ensure the integrity of sellers. One difficulty with reputation systems is that when users misbehave and their identity is tarnished they can simply apply for a new identity. Without extra infrastructure there is no way to tell whether two distinct identities represent the same entity.

Trusted platforms provide an ideal means of building more robust reputation systems. First, using trusted platforms we can ensure that a single hardware platform represents at most one identity. Consequently, to register multiple identities in a single system one would have to purchase multiple hardware platforms. This approach would thwart common attacks on reputation system where a single platform registers thousands of malicious identities. Second, trusted platforms simplify decentralized reputation systems since the platform can be used to track its own reputation.

Third-Party Computing. Increasingly computing resources are being borrowed, leased, or donated by a third party. Examples of this include (1) using donated cycles for massively parallel scientific and mathematical computations by distributed.net, SETI@home, and Folding@home, (2) using leased time on commercial computer farms for doing large-scale rendering and animation, and (3) the emerging field of grid computing that allows heavy users of scientific computing resources to pool and share their computing resources.

The difficulty with this approach to massively parallel computation is trusting the machines doing the computation to (1) produce the correct results, and (2) keep the contents of the computation secret. Trusted platforms offer an ideal mechanism for solving both problems. Using attestation, remote machines can prove that they are running the expected executable image, the trusted OS will of course keep the computation and its associated state private. The executable can use its token to sign and encrypt the results of its computation, thus ensuring its privacy and authenticity.

Civil Liberties Protection. Increasingly law enforcement requires the use of network surveillance devices [1] that can potentially infringe on civil liberties. Currently, these devices are certified not to exceed their legal boundaries by inviting a select group of experts to review their design. However, there is no guarantee that the system reviewed by the experts is the one deployed in the field. Attestation enables us to do precisely that. Building such devices on a trusted platform enables the platform to prove to third parties that the software on the device is the one authorized to execute. Note that our threat model excludes compromise of the underlying tamper-resistant hardware, which is possibly not beyond the reach of law enforcement agencies.

5 Related Work

The basic mechanisms of attestation have been well studied. Gasser et al. [13] describes an architecture which performs a secure loading process with minimal hardware support to certify to a remote party the operating systems and applications on a platform. Work by Tygar et al. [18] describes host integrity checking with secure coprocessor. More recent work by Arbaugh [6] presents a practical architecture for secure bootstrapping that provides a similar chain of integrity checks to those required for attestation. However, it is important to note that secure bootstrapping and attestation are fundamentally different capabilities. Secure bootstrapping limits what software can be run on a platform, whereas attestation merely reports what software a platform is running.

Prior work has studied attestation in a relatively limited context, usually allowing hosts within an administrative domain to certify what OS they are running

to their peers or an administrator [19]. Our much broader vision of trusted computing coincides with efforts such as TCPA [5] and Microsoft's Next Generation Secure Computing Base (NGSCB) project (formerly "Palladium") [8, 3] to deploy trusted computing platforms ubiquitously and provide a very general mechanism for application designers.

TCPA is a platform specification developed by an industry consortium to provide hardware support for trusted computing. Several current implementations of the initial TCPA 1.1b spec have already been implemented in single chips and shipped in the IBM T30 laptops. TCPA does not provide a complete solution to building a trusted platform as it deals strictly with the problem of key management and attestation. Other features required to support a flexible trusted OS (e.g. efficient architecture support for virtualization, additional protection mechanisms, a trusted path to the trusted OS, etc.) are not provided by TCPA. The software model assumed by TCPA is implicitly one of a trusted version of today's commodity OS's. As we have argued this approach is incompatible with the assurance requirements of a trusted platform.

Microsoft's NGSCB project aims to provide hardware [12] and OS support for running authenticated software on an open platform. In NGSCB trusted applications are built on top of a single dedicated trusted operating system specified by Microsoft. This operating system is protected from Windows (but not vice-versa) via special purpose hardware memory protection. All applications are limited to using only this common operating system. In short, NGSCB as it is currently described provides less flexibility and weaker isolation properties than our proposed architecture.

6 Conclusion

We have just begun to explore the broad range of potential benefits that trusted computing can bring to distributed systems. Extending trust from the core of the network to its end-points solves or greatly simplifies many problems in distributed systems as well as enabling a wide range of new applications. In future work we will continue to study the range of systems issues that arise in implementing OS support for trusted computing as well as engage in further study of its applications.

Acknowledgments

The authors would like to thank Ben Pfaff for his editorial assistance and thoughtful comments on this work. This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and through support from the Packard Foundation.

References

[1] Fbi. carnivore diagnostic tool. `http://www.fbi.gov/hq/lab/carnivore/carnivore.htm`.

[2] The mojonation p2p platform. `http://www.mojonation.net`.

[3] Microsoft next-generation secure computing base—technical FAQ. `http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/%news/NGSCB.asp`, February 2003.

[4] M. Abadi, M. Burrows, M. Manasse, and E. Wobber. Moderately hard, memory-bound functions. In *NDSS 2003*, february 2003.

[5] Trusted Computing Platform Allaince. Tcpa main specification v. 1.1b. `http://www.trustedcomputing.org/`.

[6] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *In Proceedings 1997 IEEE Symposium on Security*, pages 65–71, May 1997.

[7] Steven M. Bellovin. Distributed firewalls. *;login:*, 24(Security), November 1999.

[8] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft palladium: A business overview. `http://www.microsoft.com/PressPass/features/2002/jul02/0724palladiumwp.asp`, August 2002.

[9] N. Daswani and H. Garcia-Molina. Query-flood dos attacks in gnutella. In *ACM Conference on Computer and Communications Security*, nov 2002.

[10] Drew Dean and Adam Stubblefield. Using client puzzles to protect tls. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[11] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proc. of Crypto 92*, pages 139–147, 1992.

[12] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *Proc. of the 7th Australian Conference on Information Security and Privacy*, pages 346–361, 2002. Springer-Verlag Lecture Notes on Computer Science.

[13] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[14] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.

[15] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[16] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *Proceedings of NDSS 99*, pages 151–165, 1999.

[17] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A retrospective on the VAX VMM security kernel. In *IEEE Transactions on Software Engineering*, November 1991.

[18] J. D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[19] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, 1994.

Sensing User Intention and Context for Energy Management

Angela B. Dalton and Carla S. Ellis
Department of Computer Science
Duke University
Durham, NC, 27708
angela,carla@cs.duke.edu

Abstract

Sensors are emerging as a key area of interest in operating systems research, with a main focus on sensor networks. Turning the relationship around, we propose the use of low-power sensors as tools for improving OS-based energy management. Using sensors to detect user intent and physical context we can more directly match system I/O behavior to user needs. FaceOff is a prototype display power management system designed as a test bed and proof-of-concept.

1 Introduction

Managing energy as a resource is key to the future ubiquity of mobile computing systems. Reducing power consumption is also a major challenge in the design of mobile systems that extends beyond advances in battery technology and low-power circuit design. Energy efficient computer systems have broad environmental and economic implications [4, 5, 16]. *This position paper focuses on using sensors to leverage physical context and user intent to reduce a system's energy consumption.* We illustrate this idea with a case study on managing the display.

System level energy management approaches are currently tied almost exclusively to process execution. We believe there is ample opportunity for reducing a system's energy consumption by more directly matching the system's I/O behavior to the user's own behavior. Consider the display, a component that presents unique difficulties for power management and typically represents the largest power consumer after the CPU [9, 15]. The display exists solely for the purpose of user interaction and therefore it is only necessary when someone is looking at it. There are many times when a user may turn his attention away from a computer display, perhaps to answer a phone call or get a cup of coffee. There are also scenarios in which the display is only used briefly or not at all for a particular application. For example, someone using a computing device to play music may only interact with the device to select a song. Each time a song is selected, the user interaction would cause a timeout-based display power management scheme to reinitialize the timer. Similarly, someone may use a Personal Digital Assistant as a travel alarm clock. When the alarm sounds, there may be no need to look at the display, only to press a button in acknowledgement and turn off the alarm. However, the PDA display is turned on by the user pressing a button and would remain on for a timeout period. In these cases, turning off the display immediately or never turning it on rather than waiting for a timeout period would reduce energy consumption.

On the other hand the conventional timeout scheme which is based on lack of user input may be too aggressive for some applications. A user reading an electronic book or examining a web page with complex content might experience the annoying behavior of the display timing out. The same problem exists for a user watching a video or an automated slide show. These are situations where the user interaction is dependent on the display but is not tied to user interface input events.

In the next section we introduce our case study of a sensing system used for power management. We then describe the architecture of the system and the design of our prototype. In Section 4 we present energy measurements of our prototype to justify its potential. This is followed by a section speculating on additional roles of sensors. Section 7 outlines future work while section 6 discusses related work.

2 Architecture

As a case study we evaluate a power management method that uses a web cam mounted to the display of a laptop as a sensor. The camera periodically captures images and a face detection algorithm determines the presence or absence of a user looking at the display. Our

research is an initial investigation into using sensors combined with computer vision techniques to enhance display power management. There were several open questions we hoped to address with our evaluation. First, computer vision in general is extremely computationally intensive. Are there optimizations we can use based on the constraints of the specific problem that will make the computing costs small enough to justify? Similarly, can the optimized algorithm run quickly enough to appear seamless to the user? Can this method produce a measurable reduction of energy consumption in the system even after accounting for the added computing energy costs and the energy consumed by the camera? Finally, are there specific applications to which this method is particularly well-suited and what are the situations when it is not appropriate?

We are designing a display power management system called FaceOff as a proof-of-concept and a test bed for taking energy measurements as well as obtaining more subjective user feedback. The FaceOff architecture is simple and leaves significant room for optimizations to maximize energy savings.

The FaceOff design consists, on a high level, of three main components: image capture, face detection, and display power state control (see Figure 1). The program periodically wakes up and calls the image capture component. The image capture mechanism obtains a still image from a camera and sends the image to the face detector for analysis. The face detector

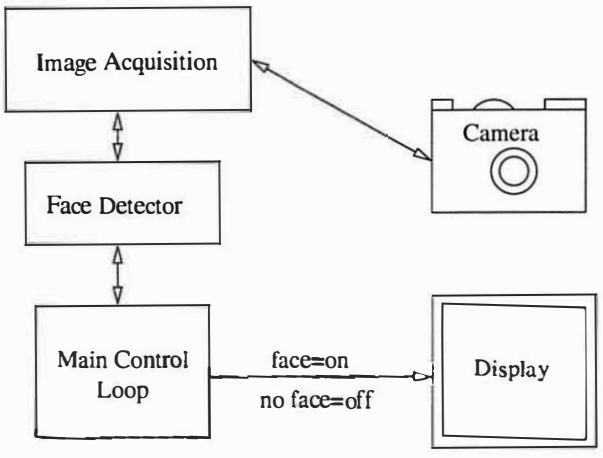


Figure 1: A diagram showing the components of the FaceOff Display Power Management System.

processes the image using an optimized algorithm and returns the Boolean value of true if a face is detected and false if no face is detected. The display power is controlled using the ACPI (www.acpi.info) interface to change the video device power state. When no face is detected the program sets the device power state of the video to a sleep state.

3 Initial Prototype

We are building the FaceOff prototype on an IBM T21 Thinkpad running Red Hat Linux. The camera is a color Logitech QuickCam 3000 web cam that connects via USB to the laptop with an average measured power consumption of 1.5W. The display power states are defined in the ACPI specification and supported by both the laptop hardware and the operating system. On this laptop, the display consumes approximately 8.5W.

Our FaceOff prototype consists of a loop that captures an image from the camera once a second. The image is saved to a file that is processed by the face detection module. An obvious optimization to the prototype is to eliminate use of the disk for storing the image acquired from the camera. However, at this point we are logging information to the disk for later analysis. Currently, the face detection module consists of a skin color detector that looks for a large central area of skin color in the image. Skin color detection was selected as a fast and fairly simple method for the initial prototype, however more accurate fast face detection methods exist and are part of our longer term plans for the FaceOff system. We are integrating ACPI based power state control into the prototype.

4 Evaluation

In order to evaluate the potential for our display power management method to achieve energy savings, we examine usage scenarios that should benefit from this approach. We measure the energy characteristics of the system assuming a best case face detector and compare that to the energy characteristics of the default timeout-based power management scheme on the same system with a typical timeout of five minutes. Current measurements were taken from a multimeter on the laptop's power supply with the laptop's battery removed to eliminate changing effects. Voltage measurements were taken for one case to verify that the voltage remains constant throughout the experiment.

In this section we present three scenarios: a large network transfer, a long, computationally intense process, and playing an mp3 song. The scenarios were selected because they offer a comparison between FaceOff and the default display power management scheme in which the timeout intuitively does not capture

the user's behavior well. It is likely that a user might turn away from the computer after initiating a large file transfer, beginning a large compilation, or starting to play a song. The FaceOff system immediately turns off the display, however it continues to perform the image polling and analysis, turning on the display when the user returns. The default system will not turn the display off until the timeout period expires, but it does not have the disadvantage of the FaceOff system overhead.

The first set of experiments measured the energy consumption of the laptop during a large network transfer. The transfers were performed using a wireless network adapter on an internal network with no other concurrent traffic. The measurements were taken assuming the best case in which the user would initiate the transfer and look away, returning as soon as the transfer completed. This application represents the case where FaceOff overhead is not expected to affect performance. We measured the total energy consumed during the tests as well as the time the network transfer took to complete. The FaceOff technique used an average of 29.5% less energy than the default, showing a significant improvement. Table 1 shows a comparison of the energy and time characteristics. Figure 2 shows traces of the power over time for one run each of the network transfer with and without FaceOff.

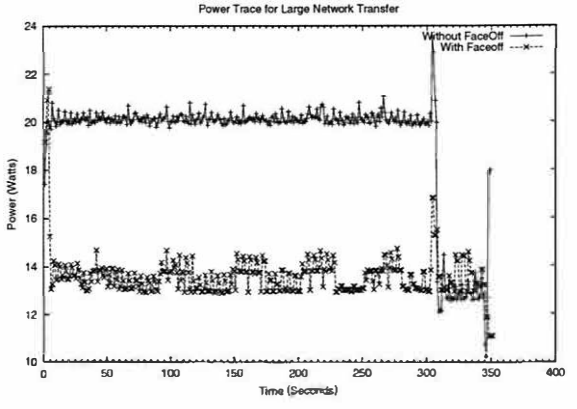


Figure 2: Power traces for large network transfer.

The second experiment measured the energy consumption of a laptop performing a computationally intensive task. In this case, the task was to compile the Linux kernel. No other programs were running on the machine except in the case of the FaceOff measurement the FaceOff prototype was running. This captures the competition for resources imposed by FaceOff. Again, we assumed the best case of the user initiating the compilation, leaving and returning immediately upon completion. The FaceOff technique resulted in an average power savings of nearly 12%. The results of the experiment reflect the fact that the default timeout-based power management system turned off the display close to halfway through the compilation, reducing overall energy consumption. Figure 3 shows sample traces of the power over time for the compilation process with and without FaceOff. The increase in completion time using each method for both experiments was insignificant.

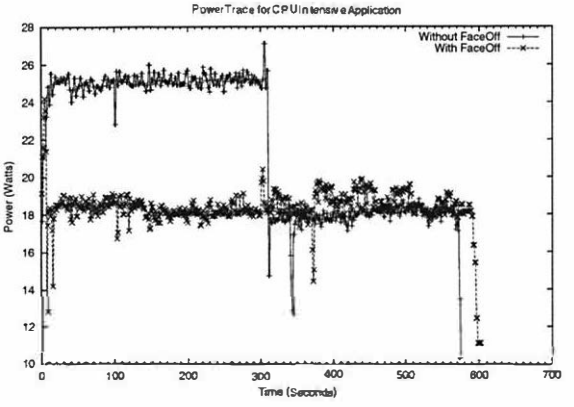


Figure 3: Power traces for Linux kernel compilation.

The third experiment, playing an mp3 song, was primarily a validation that FaceOff would cause no noticeable effect on the playback of the song. In addition, such a scenario highlights one in which the default timeout mechanism will never cause the display to turn off. The song we used in the experiment lasted 4 minutes and 11 seconds, and played with no noticeable effect when the FaceOff prototype was running. The average energy used in the default case was 4,714 Joules, versus 3,403 Joules with FaceOff, a 28% energy savings.

While the experiments we have presented provide a basis for our argument of using context awareness and user intent for power management, we believe that technological trends provide further weight in our favor. The web cam we used for the prototype system requires more power than we would anticipate a camera in any production version of the system to need. Low power, tiny CMOS cameras are now available that can be embedded into computer systems and consume as little as 20mW maximum power[7, 11]. Compared with the 1.5W average power consumption we measured for the prototype's web cam, clearly the overhead can be much lower.

Network Transfer	FaceOff	Timeout
Time (s)	351.3	348.6
Energy (J)	4791.2	6795.4
Kernel Compile	FaceOff	Timeout
Time (s)	603.5	575
Energy (J)	11023.7	12506.85

Table 1: Energy and Time Comparisons

5 More Roles for Sensors

Although our initial measurements show that the possibility exists to save energy using our method of power management, the method has significant overhead taking away from the benefits. Most notably, as realized in the initial prototype, the image capture and face detection are continuing costs, whereas an extended idle period incurs no overhead once the timeout expires.

The first observation is that people radiate thermal energy and are detectible with small motion sensors. If no person is present, no face will be detectable and therefore we do not need to capture images or run the face detection computation. Extremely small, low power pyro-electric sensors are available that can detect even slight human motion [2]. Integrating such a sensor into our system would allow the camera to be powered down until movement triggers the sensor. A conservative approach designed to minimize the delay waiting for the display to turn on would immediately turn on the display and capture an image. The face detection method would then take over until no face is detected and no motion is present. A similar optimization would be to use a touch sensor in the laptop wristrest or on the edges of a handheld device. The observation in this case is that if a user's hands are on the keyboard or holding a PDA in a particular orientation, even with no input, the user is most likely looking at the display. We can therefore either reduce the frequency of image capture or eliminate it completely and keep the display on. Face detection could also be completely suspended when there are frequent user interface events (i.e., in a sense, merging with the traditional approach).

While the main focus of discussion in this paper is display power management, there are other opportunities in which sensing context could be used for system level energy management. For example, microphones could be used to determine background noise level and possibly adjust speaker volume. Sensors could be used to determine whether to completely turn off the speakers if, for example, nobody is around to hear them. Sensing 802.11 signal strength could be used to determine whether to offload computation to a server. Remote process execution has been shown to significantly reduce energy consumption of mobile devices [13].

6 Future Work

The FaceOff prototype is a framework for applying user intent sensing to display power management. We would like to expand its capabilities to include motion sensing and enhance the face detection module, possibly adding face or gaze tracking at various levels of detail. Our aim is to evaluate what is needed to improve the accuracy of the power states while minimizing the system overhead. We plan to experiment with the frequency of image sampling to make the system more responsive during the times when motion is detected and decrease or halt the image sampling when no motion is detected. We eventually plan to incorporate a light sensor into the prototype for determining optimal display brightness.

We also plan to use the prototype to evaluate the user's experience, generally a qualitative measure rather than a quantitative one. For this reason we are examining ways of quantifying the user's experience with the prototype, for example by adding a button to indicate annoyance at the display state changes. We would like to be able to gauge the accuracy of the prototype in determining context.

We plan to do a comprehensive user study characterizing usage patterns similar to the study done in [8]. We will use the study to provide estimates of energy savings from our display power management technique under realistic laptop workloads. In addition to capturing usage patterns for laptops, which we can do using our FaceOff prototype, we would like to study other mobile devices that could benefit from our system.

7 Related Work

As power management at a software level has gained attention both in research and industry, several standards have emerged. The first standard was *Advanced Power Management (APM)*, a BIOS-based power management specification. APM provides CPU and device power management. Device power states are transitioned based on timeouts. Problems found as APM matured led to the development of the *Advanced Configuration and Power Interface (ACPI) Specification*. In ACPI, power management decisions are made by the operating system rather than the BIOS [9]. Both APM and ACPI provide an interface for changing the power state of the display through software using *Display*

Power Management Signaling (DPMS). APM and ACPI provide hooks to manage the power state of the display, however currently the only strategy for taking advantage of the lower power states is turning off the backlight and display after a certain period of time with no user input [10]. The Compaq iPAQ PocketPC has an additional method of display power management using an ambient light sensor to allow for adaptable display brightness. The only other novel policy ideas we found for reducing the power consumption of the display were zoned backlighting proposed by Flinn and Satyanarayanan, a method which presupposes hardware that is not yet available [5], and the recent work on Energy-Adaptive Display System Designs [8] that proposes software optimizations called *dark windows.*

There are several projects that involve face detection and face, gaze and eye tracking for perceptual user unterfaces. The Smart Kiosk System uses vision to detect potential users and decide whether the person is a good candidate for interaction. [12]. CAMSHIFT is a face tracker that is being used to control games and 3D graphics by defining head movements to perform specific actions [1]. Another related project is a perceptual user interface for recognizing predefined head gesture acknowledgements. The face detection is performed by using an IBM PupilCam to locate the pupils in the image and then uses simple image processing techniques to detect the upper face region [3]. A series of articles on Attentive User Interfaces discusses several projects that use eye tracking to design context-based user interfaces [14]. To our knowledge there are no other projects that are integrating face detection and power management.

8 Conclusion

Sensors as components of sensor networks have recently become an interesting target domain for operating systems research (e.g., TinyOS [6]). In this position paper, we turn this around and consider low-power sensors as tools in the service of OS-based energy management for mobile computers. As a case study, we consider sensors providing information from which to infer user intention and user context as it affects energy management of the display – capturing the direct dependency that looking at the screen suggests a need for it to be illuminated. Intuitively, this is attractive as a more direct indication of the user's need for display power consumption than the keyboard and mouse input events used in traditional timeout-based strategies.

Our preliminary exploration of this use of sensors to inform the OS combines currently available technology that allows software to switch the display power state, low-power sensors, and face detection techniques. We have proposed a method of reducing display power consumption by turning the display off in the absence of a user. Face detection, while a computationally intensive technique, can be optimized for the simplified problem of detecting an upright, frontal face of an approximate size indicating the presence of a user looking at the display. For our FaceOff prototype, a web cam acquires images and the computer's own CPU performs skin color based face detection.

Measurements of power consumption using the prototype system indicate the promise of significant energy savings from this type of context-based display power management scheme. Camera technology trends indicate that cheap, very low power cameras are becoming more readily available and could produce greater net energy savings in the future using our technique. Furthermore, the specific task of user detection could potentially be optimized using additional low power sensors combined with less computationally intensive techniques to further reduce overall energy consumption.

While the most obvious immediate benefit of our display power management system would be extending the battery life of mobile devices, the method could also provide the basis for energy savings on larger scale displays. One interesting possibility is to apply a similar technique to very large displays, using gaze tracking to determine what part of the display to turn on.

In this position paper, we have demonstrated that exploiting low-power sensors to assist the OS in inferring user intention and context for the benefit of energy management is a fruitful direction.

9 Acknowledgments

This work is supported in part by the National Science Foundation (ITR-0082914,CCR-0204367).

References

[1] Gary R. Bradski. Computer vision face tracking for use in a perceptual user interface. *Intel Technology Journal*, 2(2):12–21, 1998.

[2] Glolab Corporation. Parts for pyroelectric infrared motion detector. http://www.glolab.com/pirparts/pirparts.html.

[3] James W. Davis and Serge Vaks. A perceptual user interface for recognizing head gesture acknowl-

edgements. In *Workshop on Perceptive User Interfaces 2001*, Orlando, FL, 2001.

[4] C. S. Ellis. The Case for Higher-Level Power Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999.

[5] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, October 2000.

[7] Wilderness Security Inc. Bd-127 high resolution cmos camera. http://www.wildernesssecurity.com/bd127.htm.

[8] Subu Iyer, Lu Luo, Robert Mayo, and Parthasarathy Ranganathan. Energy-adaptive display system designs for future mobile environments. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services*, May 2003.

[9] J. Kolinski, B. Press, and A. Henroid. *Power Management History and Motivation*, chapter 2, pages 7–17. Intel Press, September 2001.

[10] Jacob Lorch and Alan J. Smith. Software Strategies for Portable Computer Energy Management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.

[11] Space Television Ltd. Color and b/w cmos board camera. http://www.spacetv.co.za/YOKO/PG22-23.htm.

[12] J. Rehg, M. Loughlin, and K. Waters. Vision for a smart kiosk. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 690–696, 1997.

[13] Alexey Rudenko, Peter L. Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. The remote processing framework for portable computer power saving. In *Selected Areas in Cryptography*, pages 365–372, 1999.

[14] Roel Vertegaal. Attentive user interfaces. *Communications of the ACM*, 46(3), 2003.

[15] Gregory F. Welch. A survey of power management techniques in mobile computing operating systems. *Operating Systems Review*, 29(4):47–56, 1995.

[16] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

Access Control to Information in Pervasive Computing Environments

Urs Hengartner and Peter Steenkiste
Carnegie Mellon University
{uhengart,prs}@cs.cmu.edu

Abstract

Many types of information available in a pervasive computing environment, such as people location information, should be accessible only by a limited set of people. Some properties of the information raise unique challenges for the design of an access control mechanism: Information can emanate from more than one source, it might change its nature or granularity before reaching its final receiver, and it can flow through nodes administrated by different entities. We propose three design principles for the architecture of an access control mechanism: (1) extract pieces of information in raw data streams early, (2) define policies controlling access at the information level, and (3) exploit information relationships for access control. We describe an example architecture in which we apply these principles. We also report how our earlier work about adding access control to a people location service contributed to the more general access control architecture proposed here.

1 Introduction

Pervasive computing environments, such as the ones studied in CMU's Aura project [4], provide many kinds of information. Some of this information should be accessible only by a limited set of people. For example, a person's location is a sensitive piece of information, and releasing it to unauthorized entities might pose security and privacy risks. For instance, when walking home at night, a person will want to limit the risk of being robbed, and only people trusted by the person should be able to learn about her current location.

The access control requirements of information available in a pervasive computing environment have not been thoroughly studied. This information is inherently different from information such as files stored in a file system or objects stored in a database, whose access control requirements have been widely studied. In this paper, we discuss these differences in detail. This discussion leads to the proposal of three design principles for the architecture of an access control mechanism deployed in a pervasive computing environment. Namely, these design principles suggest: (1) extract pieces of information in raw data streams early, (2) define policies controlling access to information at the information level, and (3) exploit information relationships when performing access control.

Let us illustrate the unique challenges for access control with an example scenario. Figure 1 shows how individuals can locate other people and free food in a pervasive computing environment. To simplify our discussion, we introduce the following notation: We call entities from which original information emanates *source nodes* (file system, access point, camera, and instant messaging client in the example), and entities that are final recipients of information *sink nodes* (Bob). For a pair of entities between which there is a direct information flow (e.g., laptop locator to people locator and camera to face detector), we call the entity from which the information emanates *server node* and the entity receiving the information *client node*. Each source node is always a server node, and each sink node is always a client node. The notation {foo; bar} denotes that a server node reports information "bar" about item "foo" to a client node, whereas the item is optional.

Based on Figure 1, we identify the following challenges for performing access control to information in a pervasive computing environment:

- A piece of information, like a person's location, can be derived from other pieces of information, like a camera picture or the location of her laptop, and multiple source nodes can contribute to a single piece of information. The source nodes can be of different types: they can be devices like sensors or cameras, or they can be of digital nature like files or instant messages. Thus there might be no single point, like a file in a filesystem or an object in a database, where we can deploy access control to protect the information. Instead, we have to run access control in a distributed way, depending on how

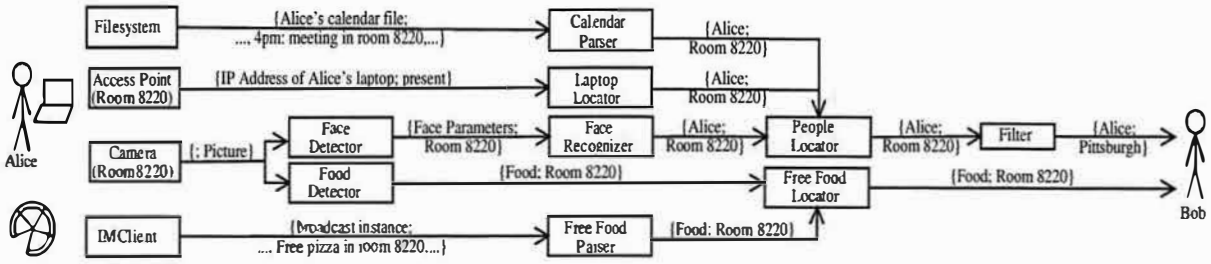


Figure 1: *Retrieval of location information about people and free food. The people locator gathers people location information from various nodes. The calendar parser extracts scheduling information from a user's calendar file. The laptop locator returns the location of the wireless access point to which a user's laptop is connecting. The face recognizer maps between a person and a set of face parameters used by the face detector. The face detector detects faces in pictures delivered to it by cameras in a building. The free food locator locates free food by exploiting information from the food detector and the free food parser. The food detector detects food in camera pictures. The free food parser scans broadcast messages posted to an instant messaging application for messages announcing free food. The filter reduces the granularity of Alice's location information before giving it to Bob.*

information is gathered and processed.

- Information can change in nature (e.g., a picture becomes becomes a user/location pair) or its granularity (e.g., "Room 8220" becomes "Pittsburgh") while flowing from the source to the sink node. The access control mechanism needs to be aware of these changes. For instance, it should be possible to grant only a small set of people access to a piece of information provided at a fine-grained level, whereas a larger set can have access to the same information provided at a more coarse-grained level. Access control to files or objects in a database covers only a single variant of a piece of information, being of a single granularity only.

- The nodes in the environment can be administrated by different entities. Thus access control needs to ensure that information flows only through nodes that are authorized to access the information. A filesystem or a database is typically run by a single administrative entity, and accessing remote filesystems from outside that domain, as, for example, in the case of the Andrew File System (AFS), requires extra configuration beforehand.

In addition to addressing these challenges, the access control infrastructure needs to offer flexible ways for granting entities access to information. Though this flexibility could also be useful for information like files or objects in a database, it becomes more important for information provided in a pervasive computing environment. As mentioned above, we should be able to grant access based on the granularity of information. We also require flexibility along the following lines:

- We should be able to grant access to an individual based on her context. An individual's context can consist of the current time, her current location, or her current activity. For example, Alice can allow Bob to access her location only during office hours.

- Depending on the environment, different entities should be able to grant access to information. For example, in a military environment, a central authority must perform this task, whereas in a university environment, individuals should be able to grant access to their personal information.

- Individuals should be able to let other people grant access for them. For example, an individual can have her doctor decide who should have access to her medical information, such as a sensor measuring her heart rate.

In the rest of this paper, we elaborate on how these challenges and requirements affect the design of an access control architecture deployed in a pervasive computing environment. In Section 2, we introduce three design principles for such an architecture. In Section 3, we present the design of our actual architecture. In Section 4, we discuss how our experience gained in designing and implementing an access control mechanism for a people location service contributed to the proposed design principles and architecture. In Section 5, we discuss related work. We conclude the paper in Section 6.

2 Design Principles

In this section, we elaborate on three principles that we have followed in our design of an access control archi-

tecture for information available in a pervasive computing environment.

2.1 Extract Information Early

This principle suggests that for raw data streams, such as a videostream, the pieces of information available in the stream should be extracted as early as possible. Access to the raw data streams should be strictly limited and granted only to the entity doing the extraction. For example, when exploiting cameras for locating people or free food, only the face and food recognizers should have access to the pictures delivered by the cameras. As soon as information has been extracted from a raw data stream, more entities can be granted access to the extracted information. However, these entities should not get access to the raw data stream, which has a lot of other information in it. For example, a person can be granted access to her location information as produced by the face recognizer, but not to the raw pictures. Similarly, each person in the organization can be granted access to the location of free food as produced by the free food parser, but not to the raw pictures.

2.2 Define Policies at Information Level

This principle suggests that users should not have to issue policies controlling access to information at the level of individual nodes. Instead, they should be able to define these policies at the information level. The principle is based on the observation that a pervasive computing environment consists of a lot of nodes. Therefore, the environment is going to require a lot of policies to control what nodes should have access to what information offered by other nodes. To reduce the burden on individual users, we do not have them issue low-level statements about nodes. Instead, we let them make high-level statements about information and relationships between information. For example, Alice should be able to state that "Bob can access my location information" and "Use the location of my laptop for locating me". She should not have to declare that "Bob can access my location information as provided by the people locator, which is derived from the location of my laptop as provided by the laptop locator" and "The people locator can access the location of my laptop as provided by the laptop locator". The high-level statements should be independent of the information flow, that is, users should not have to know about the architecture of the system and how information flows through the various nodes.

2.3 Exploit Information Relationships

This principle suggests that information relationships, as proposed by the second design principle, should be taken into account for access control. Such relationships can be exploited in different ways. We describe two examples. First, if there is a relationship specifying that the location of Alice's laptop should be used for locating her, the laptop locator will grant the people locator access to location information about Alice's laptop. This example is a straightforward mapping from a high-level information relationship to a low-level access control policy. Second, we can require that the people locator proves to the laptop locator that Bob has actually asked for the location of Alice and that Bob is entitled to get this information. This additional constraint implies that the people locator will not be able to ask the laptop locator for Alice's location unless it can present a valid request for the location of Alice. Therefore, we minimize damage in case of a break-in into the people locator; intruders will not be able to ask the laptop locator for the location of Alice since they are not able to generate a valid request for Alice's location (unless they explicitly have the right to generate such a request). This example shows that by exploiting additional knowledge available in an information relationship, we can make the access control process more robust.

3 Design of Access Control Mechanism

Before forwarding a piece of information from a server to a client node, we need to validate that the client node is entitled to get the information. Two possible models for performing access control are end-to-end and step-by-step. In this section, we first discuss these two models and their application in the design of our access control architecture. We then describe our architecture.

3.1 End-to-End vs. Step-by-Step Model

In the end-to-end model, a source node validates that the sink node and all the nodes between the source and the sink node are authorized to receive the requested piece of information emanating from the source node. The source node then instructs all the nodes on the path to the sink node how to deal with the received information; these instructions can be sent together with the information. The advantage of this model is that access control is done at a single point; there are no redundant access control checks. The drawback of the model is that it puts a heavy load on a source node. The task of performing access control might be too heavyweight for resource-limited source nodes, such as a sensor. In addition, there might be multiple source nodes, so there is no single point at which we can perform access control. Another drawback is that a request for information has to flow through the entire system to the source node before an access control decision is made. Intermediate nodes process the request and potentially translate it

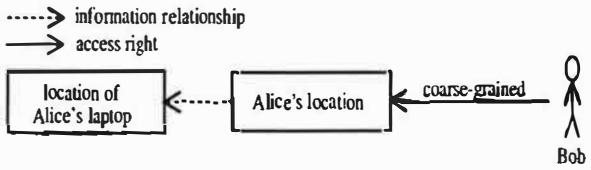


Figure 2: *Example information relationship and access right. Alice's location information is derived from the location of her laptop. Bob has coarse-grained access to Alice's location information.*

into different requests. Therefore, the end-to-end model is more prone to denial-of-service attacks.

In the step-by-step model, for each possible pair of server/client nodes participating in the information flow, the server node validates whether it should give information to the client node. The advantage of this model is that it distributes the access control load over multiple nodes. In addition, an invalid request can be thrown away immediately by the first node receiving the request, thus this method is less prone to denial-of-service attacks. The drawback of this model is that all nodes need to run access control, thus there might be redundant checks. In addition, all nodes need to be able to run access control checks. This requirement is difficult to fulfill when we want to support "dumb" nodes with limited functionality. For example, the filter node shown in Figure 1 might only be able to filter data, but not to run access control.

Our architecture is based on the step-by-step model. We extend the model to support dumb nodes by having them delegate the access control check to a node that is able to perform access control.

Let us now discuss how we apply this access control architecture in the example scenario shown in Figure 1. We limit our discussion to a subset of the nodes; the remaining nodes run access control in a similar way. The nodes uses the information relationship rule and the access right associated with Alice's location shown in Figure 2. Note that the figure does not determine the identity of the entity making these associations. Depending on the environment, this entity can be a central authority, or it can be Alice, whereas Alice can delegate this task to some other entity. Also note that we have to ensure that only valid information relationships are exploited in the access control process. For example, Alice should not be able to state that Bob's laptop should be used for locating her.

The nodes perform access control as follows: The filter is not able to perform any access control. It only filters location data as instructed by the people locator, and has the people locator take care of access control. The people locator validates that Bob has access to Alice's location information and at what granularity. It will instruct the filter to reduce the granularity of the information accordingly. It requests location information from the various location services, for example, from the laptop locator. The laptop locator validates that the location of Alice's laptop should be used to determine Alice's location by looking at the information relationship. In addition, as suggested by the third design principle, it can verify whether Bob has really asked for Alice's location information. The access point is a resource-constrained node: its access control ensures only that information is forwarded to the laptop locator, but not to any other nodes.

3.2 Architecture

Our access control architecture consists of two description languages and the actual access control mechanism.

We use the *policy description language* for the specification of policies controlling access to information. A policy lists the entities that are granted access to a particular piece of information. Policies, and thus the description language, should be flexible. For instance, it should be possible to grant access both to single entities and to groups of entities. In addition, policies should support the specification of various constraints (e.g., time intervals) that need to be fulfilled before access is granted. Also, it must be possible to specify the granularity of a piece of information to which access is given. Depending on the environment, different entities should be able to define these policies. The entity can be an individual or a central authority. For some environments, the individuals should have the option to delegate the decision about her access control policies to other entities. We can employ existing certificate-based trust management frameworks (e.g., KeyNote [2] or SDSI/SPKI [3]) to implement this component.

We use the *information description language* to describe information, its properties, and relationships between information. For example, Alice needs to be able to specify the granularity of information to give granularity-based access rights. In addition, if Alice wants her laptop's location to serve as her location, she needs to be able to explicitly define this relationship. Finally, if there are multiple services that provide location information, Alice must be able to define which of them provides her location information.

At each server node participating in an information flow, the access control mechanism decides whether information should flow to the corresponding client node. To make this decision, the access control mechanism will use the policy and information description languages just described. More specifically, the access control mechanism works as follows.

First, the mechanism checks whether there is a policy allowing the information flow from the server node to the client node. In addition, all the constraints stated in the policy must be fulfilled, and the granularity of the information might have to be reduced. The policy description language lets the access control mechanism learn about the content of policies.

Second, the mechanism also exploits information relationships in the access control process, as suggested by the third design principle. Furthermore, it can exploit additional properties of the information. For example, anyone who has access to a piece of information offered at a fine-grained level should also have access to the same piece of information offered at a coarse-grained level. In addition, the mechanism might be able to automatically derive the access control policy of a piece of information derived from other pieces of information. Namely, it can combine the access control policies of these pieces to form the access control policy of the derived piece. For the combination, we can, for example, exploit principles from Myers and Liskov's decentralized label model [10]. The information description language lets the access control mechanism learn about the characteristics of information.

Finally, to avoid snooping attacks on information exchanged between nodes, the mechanism needs to encrypt this information. We cannot employ end-to-end encryption since intermediate nodes have to be able to process the information flowing through them.

4 People Locator Service

We have built a service for locating people and are currently deploying it at Carnegie Mellon [5]. The service uses multiple sources for locating people, namely, login information, calendar information, and device location information. We have included access control in the design of our service, and the experience gathered during its development and implementation has contributed to the architecture proposed in this paper.

We use SPKI/SDSI digital certificates [3] for specifying different types of decisions, such as policies controlling access to information, and performing the access control check. For example, in the following certificate, Alice grants Bob access to her location information at a coarse-grained level only. Bob can locate Alice only if she is at the locations specified in the certificate and during the listed time intervals. (The signature belonging to the certificate is not shown.)

```
(cert   (issuer (publicJkey_of_alice))
        (subject (public_key_of_bob))
        (tag (policy alice
           (* prefix world.cmu.wean)
           (* set  (monday (* range numeric
                        ge #8000# le #1200#))
                   (tuesday (* range numeric
                        ge #1300# le #1400#)))
           coarse-grained)))
```

The example shows that we can define flexible access control policies in the tag section of a certificate. Therefore, we are planning on employing SPKI/SDSI certificates in the more general access control architecture. But we have also identified a limitation of this mechanism. The fixed sequential ordering of the access control constraints in the tag section limits the type of constraints that users can specify. It is not possible for users to include additional, individual constraints (such as customized filters) in their certificates.

Clients of our service contact the people locator node. To limit risk in case of a break-in, we do not allow this node to actively ask the various source nodes for information. Instead, it can only forward requests received from its clients to them. These nodes return information to the people locator node only if the entity setting up the access control policy for the requested information trusts this node. While this mechanism is sufficient for the people locator service, it is not flexible enough for a more general system. Forwarding requests assumes that each node provides the same type of information; it does not support the case where nodes offer different types of information (e.g., the location of a device instead of the location of a person). In addition, the notion of trust is rather awkward since it is not directly coupled to the information to which access is controlled. Instead, it is coupled to nodes. These shortcomings lead us to propose the second and third design principle.

5 Related Work

Previous work about dealing with intermediate nodes between a source and a sink node includes, for example, Howell and Kotz's quoting gateways [6] and Neuman's proxy-based authorization [11]. In this related work, there is typically only one intermediate node between the source and the sink node, whereas in a pervasive computing environment, there can be an unlimited number of nodes. In addition, in the related work, the

intermediate nodes are authorized to ask for data themselves. This authorization presents a risk if the intermediate node is broken into. We try to avoid such risks by exploiting opportunities offered to us by the information description language.

Kagal et al. [8] propose an extended role-based access control model for pervasive computing. (Al-Muhtadi et al. [1] pursue a similar approach.) They extend the model with the notion of delegation and incorporate context-sensitive roles. The authors use the model for controlling access to services like a printer or a projector. In contrast to our work, the authors do not consider preventing individuals from accessing pieces of information provided by a service. For access control, the proposed architecture relies on a centralized trusted entity running a Prolog-like knowledge base. Our architecture runs access control in a distributed way.

Jiang and Landay [7] and Minami and Kotz [9] perform access control to information by assigning tags to pieces of information. A tag denotes the policy for the piece. The tag of a piece of information derived from other pieces of information is automatically computed based on the tags of these pieces. Whereas Jiang and Landay use a technique similar to Myers and Liskov's decentralized label model [10] for this computation, Minami and Kotz deem this model too conservative and propose a more relaxed model. Though automatic derivation of policies can reduce the number of policies that need to be manually specified in a pervasive computing environment, it is not clear how big the actual benefit is. For example, when information changes its type or becomes more coarse-grained, other people or more people than the ones allowed to access the original information might be granted access to the transformed information. In such a case, the new tag cannot be automatically derived. Finally, both Jiang and Landay and Minami and Kotz assume that the environment is administrated by a single entity and that source and intermediate nodes are fully trusted. Thus, in contrast to our model, there is no need to restrict intermediate nodes from accessing information, and the authors address only preventing sink nodes from getting access.

6 Conclusions

In this paper, we examined access control requirements for information available in pervasive computing environments. We identified several challenges, and we presented three design principles that we applied in an example access control architecture. We are currently refining our design. Namely, we are designing a language for the specification of relationships between different pieces of information and of their granularity. This language will help us in defining policies controlling access to information and in automatically deriving access control policies.

Acknowledgments

We thank Mahim Mishra and the anonymous reviewers for their comments. This research was funded in part by DARPA under contract number N66001-99-2-8918 and by NSF under award number CCR-0205266. Additional support was also provided by Intel.

References

[1] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and M. D. Mickunas. Cerberus: A Context-Aware Security Scheme for Smart Spaces. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 489–496, March 2003.

[2] M. Blaze, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704, September 1999.

[3] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693, September 1999.

[4] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, April-June 2002.

[5] U. Hengartner and P. Steenkiste. Protecting Access to People Location Information. In *Proceedings of International Conference on Security in Pervasive Computing (SPC 2003)*, March 2003.

[6] J. Howell and D. Kotz. End-to-end authorization. In *Proceedings of the 4th Symposium on Operating System Design & Implementation (OSDI 2000)*, pages 151–164, October 2000.

[7] X. Jiang and J. A. Landay. Modeling Privacy Control in Context-Aware Systems. *IEEE Pervasive Computing*, 1(3):59–63, July-September 2002.

[8] T. Kagal, L. Finin and A. Josh. Trust-Based Security in Pervasive Computing Environments. *IEEE Computer*, pages 154–157, December 2001.

[9] K. Minami and D. Kotz. Controlling access to pervasive information in the "Solar"system. Technical Report TR2002-422, Dept. of Computer Science, Dartmouth College, February 2002.

[10] A. C. Myers and B. Liskov. Complete, Safe Information Flow with Decentralized Labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998.

[11] B.C. Neuman. Proxy-Based Authorization and Accounting for Distributed Systems. In *Proceedings of International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

Privacy-Aware Location Sensor Networks

Marco Gruteser, Graham Schelle, Ashish Jain, Rick Han, and Dirk Grunwald
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309
{gruteser,schelleg,ajain,rhan,grunwald}@cs.colorado.edu

Abstract

Advances in sensor networking and location tracking technology enable location-based applications but they also create significant privacy risks. Privacy is typically addressed through privacy policies, which inform the user about a service provider's data handling practices and serve as the basis for the user's decision to release data. However, privacy policies require user interaction and offer little protection from malicious service providers. This paper addresses privacy through a distributed anonymity algorithm that is applied in a sensor network, before service providers gain access to the data. These mechanisms can provide a high degree of privacy, save service users from dealing with service providers' privacy policies, and reduce the service providers' requirements for safeguarding private information.

1 Introduction

Sensor network technology promises a vast increase in automatic data collection capabilities through efficient deployment of tiny sensing devices. Arrays of sensors could be deployed alongside roads to monitor traffic patterns or inside buildings to sense contextual information for adaptive computing services. In particular, there is great interest in location tracking systems, which determine the position of users for location-based services. We foresee that sensor network technology decreases the cost of such systems by replacing cables with multi-hop radio communications and allowing in-network processing of data.

While these technologies offer great benefits to users, they also exhibit significant potential for abuse.[1] Particularly relevant are privacy concerns, since sensor network technology provides greatly expanded data collection capabilities.

A common approach addresses privacy concerns at the database or location server layer—after data has been collected. For example, privacy policies govern who can use an individual's data for which purposes [2, 3, 4]. Furthermore, data perturbation [5] or anonymity mechanism [6, 7] provide access to data without disclosing privacy sensitive information. However, data is difficult to protect once it is stored on a system. In the past, private data has been inadvertently disclosed over the Internet and companies have distributed data in violation of their own privacy policies. In addition, data theft and distribution through company insiders poses a serious challenge. Such approaches also do not address the risks that an adversary circumvents the location server and directly collects data from the location tracking system.

This paper leverages sensor nodes' data processing capabilities to enhance privacy through distributed, in-network anonymity mechanisms. These mechanisms are applied before data leaves the sensor network and can be stored in a location server; thus, databases and locations servers are removed from the trusted computing base, meaning users only need to trust the sensor network itself. A third party, independent from the data consumers, could install and service the network to establish user trust. The paper concentrates on location sensor networks, since location information is especially privacy sensitive and potentially specific enough to reveal the identity of individuals. Specifically, the paper contributes the following key ideas:

- a discussion of privacy risks and attacks for location sensor networks
- a distributed privacy algorithm that cloaks location information to preserve anonymity
- a complimentary routing scheme and election algorithm that chooses leaders for hierarchically organized entities in physical space

2 Related Work

Privacy concerns in location-based application scenarios are typically addressed in a location broker residing in the middleware layer. To our knowledge, Spreitzer and Theimer [8] pioneered the development of such an architecture. In this work, each user owns a trusted user agent that acts as an intermediary. It collects location information from a variety of sensors and controls application access to this data.

More recent research addresses the specifics of privacy policies, on which access control decisions are based. For instance, Myles and colleagues [9] describe an ar-

[1] Indeed, at least in one case a man stalked his former girlfriend aided by a GPS device and digital cellular transmitter mounted on her car [1].

chitecture for a centralized location server that controls access from client applications through a set of validator modules that check XML-encoded application privacy policies. In the automotive telematics domain, Duri and colleagues [4] present a policy-based framework for protecting sensor information, where an in-car computer can act as a trusted agent. Hengartner and Steenkiste [10] point out that access control decisions can be governed by either room or location policies; thus, such systems should be able to resolve conflicts between several different policies. Snekkenes [3] presents advanced concepts for specifying policies in the context of a mobile phone network. These concepts enable access control based on criteria such as time of the request, location, speed, and identity of the located object. However, the author concludes by expressing doubt that the average user will specify such complex policies. In addition, privacy policies mainly serve as a vehicle for establishing trust in a service provider—they cannot guarantee that the provider adequately protects the collected data from in- or outside attacks.

Anonymity mechanisms present an alternative to privacy policy-based access control through depersonalization of data before its release. Specifically, Gruteser and Grunwald [11] analyze the feasibility of anonymizing location information for location-based services in an automotive telematics environment. In addition, Beresford and Stajano [12] independently evaluate anonymity techniques for an indoor location system based on the Active Bat. These approaches address the problem of too precise location information that enables identification of a user or continued tracking of movements. However, access control or anonymity mechanisms in the middleware offer little protection when the location tracking system (the sensors) are owned by an untrusted party, such as in a shopping mall.

The Cricket Location-Support System [13] incorporates privacy concern in the design of the location sensor system itself. The system comprises a set of beacons embedded into the environment and receiving devices that determine their location through listening for the radio and ultrasound beacons. This approach enhances user privacy over previous systems, such as the Active Badge [14] and the Active Bat [15], because device location information is initially only known to the devices themselves. The owner can then conceivably decide to whom this data should be released. Therefore, users do not need to trust the embedded sensors or a location server. However, it requires the user to carry a device that is compatible with the beacons and powerful enough to make access control decisions, to delegate them to the user (via a suitable interface), or to communicate the request to another trusted agent. It does not cover other classes of location-tracking systems, where the user carries no device (e.g., infrared cameras) or the device is not powerful enough to allow such decision-making (e.g., RFID or the Active Bat).

3 Design Considerations

One usage example of a location sensor network is an in-building occupant movement tracking system. Such a location system would be useful for architectural and interior design, since it would deliver data on the popularity and usage of different building areas such as conference rooms, alcoves, individual offices, or supermarket aisles.[2] However, employees or customers might be concerned about their privacy. We will revisit this example throughout the paper.

These applications require aggregate statistics on the popularity of certain locations but not necessarily precise information about a person's location at any given time. Therefore, we argue that this problem can reasonably be addressed through anonymity mechanisms that reduce data quality within known bounds to maintain a well-defined level of anonymity in different situations.

We do not restrict the system to a specific location sensing technology but make the following assumptions. The location tracking system comprises an array of sensor nodes, one or more base stations, and a location server. The sensor nodes are resource limited computing devices with wireless communication capabilities (e.g., [17, 18]). The sensors itself should be capable to determine the number of individuals in an area and monitor changes in real-time. Base stations bridge the wireless sensor communications into the wired network, where the location server collects the sensor data and publishes it to applications.

The sensor system periodically reports location information as a set of tuples (c, a) where a labels an area and c the count of data subjects, who visited the area during the period. Areas are hierarchically organized; therefore, the network can present an overall count for a certain area in addition to counts for smaller sub-areas within.

3.1 Privacy Threats and Attack Model

We define a location privacy threat as an instance in which an adversary can obtain an individual's (the data subject's) location information through the location system *and* can identify the individual. For example, through the location system an adversary could obtain the current position of every individual. Continuous access to this information would allow him to track movements of an unknown user. However, for this to constitute a location privacy threat, the adversary must also be able to link identities to the reported user locations.

To identify individuals, the adversary can have prior information about the people and space that are monitored. For example, knowing who owns a particular office would most likely correctly identify a person that is monitored in this office [12]. The adversary can simply

[2] In fact, the IBM Footprint research project [16] developed an inexpensive $10 infrared sensor. An array of such sensors allows stores to measure the effectiveness of their store design by tracking the path of customers through the store. For example, it reveals whether promotional items are effectively placed, whether customers stopped to look at promotions, or how long customers had to search for a specific item.

link these two pieces of information and conclude that with very high probability the identified individual is in his office. Once identified, he can then track the individual's movements to other areas of the building by monitoring the location updates. Through adaptively changing data precision, the sensor network seeks to prevent (or at least make sufficiently difficult) that an adversary can link prior information with the information obtained through the sensor system. The network should only reveal precise locations of groups of people, but not of individuals and their paths. Inspired by Samarati and Sweeney [19, 6, 7], we consider the data k-anonymous, if every location reported from the network is indistinguishable from at least k 1 other subjects.

This work also considers a more sophisticated adversary, with local access to the sensor network, who attacks the network to gain more precise location information. In particular, the adversary could mount the following attacks:

- Passive Attacks

  Eavesdropping. The adversary could simply listen to data and control traffic. Control traffic conveys information about the sensor network configuration. Data traffic contains potentially more detailed information than accessible through the location server.

  Traffic analysis. An increase in the number of transmitted packets between certain nodes could signal that a specific sensor has registered activity.

- Active Attacks

  Insert false data. A malicious node could trick the system into reducing data distortion (privacy protection) through spoofing subjects.

  Change routing behavior. An inserted or compromised node could drop packets, forward them incorrectly, or advertise itself as the best route to all nodes (blackhole effect) in an attempt to gain information.

This paper focuses on user privacy; hence, we do not consider attacks such as denial of service, where the adversary does not learn any private information.

4 System Design

A network is needed that provides near real-time location information with the properties that it preserves k-anonymity with respect to the described attack model while still delivering useful data. To achieve this goal, we take the following approach.

4.1 Approach

Data cloaking. The sensor network perturbs the sensed location data so that it meets the k-anonymity criterion. Ideally, the network applies only the minimum necessary perturbation so that the data retains its usefulness for a large number of applications.

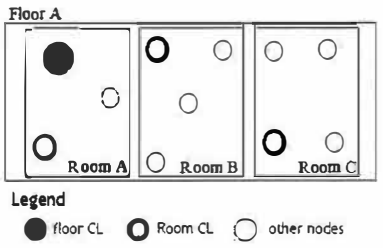


Figure 1: The desired result of coordination leader election is one CL at each hierarchical level

Hierarchical aggregation. Network nodes organize distribution of sensed location data through a spanning tree. Multiple nodes throughout the spanning tree, the coordination leaders (CL), cloak data so that no single entity has a complete view of the original data. The hierarchy should reflect the spatial characteristics of the area. For example, it could be organized into cubicles, rooms, floors, and buildings.

Secure and unobservable communications. Nodes communicate with encrypted and authenticated data packets (e.g., using the SPINS protocols [20]) to prevent eavesdropping and active attacks. In addition, data transmissions are periodic and independent from sensor readings to protect against traffic analysis.

4.2 Coordination Leader Election and Spanning Tree Construction

The decentralized cloaking and aggregation mechanism requires one coordination leader for every level of hierarchy; for example, one CL for every room, for every floor, and for the building. Figure 1 depicts such a configuration. All data flows from the individual nodes first to the room CL and then to the floor CL, which sends the data to a location server. Since CLs can be outside the single-hop radio range, network nodes need to establish routes to higher-level coordination leaders. The node routing tables will hold an entry for each of the hierarchical CLs that this node belongs to. Referring to Fig. 1, a node in Room B may be required to forward packets to its room CL, while also forwarding packets from Room C bound for the floor CL. For n levels of hierarchy, n routing table entries will be required. Notice that the size of the routing table scales with the total number of hierarchies, rather than the number of nodes in the network.

A hierarchical node ID assignment that mirrors the characteristics of the physical area simplifies coordination leader election. To this end, the ID describes where the node is physically located (e.g., in which room). The node ID is subdivided into several bitfields that determine its identification at every level within the hierarchy.

Every node will have a unique ID, but nodes within the same room will share the same room ID, while nodes on the same floor will all share the same floor ID. The IDs and the hierarchy are statically configured during system installation.

Coordination leader election and routing table setup uses a 3-way handshake protocol. The process starts with a root node, such as a base station or the location server to elect coordination leaders for the top level (e.g. floors). The selected coordination leaders than recursively apply the protocol to find CLs for their sublevels until leaders are elected for all levels. The handshake involves the three packet types CL_REQUEST, CL_REPLY, and CL_CONFIRM, which simply contain the sender and receiver node ID, a hop count, and the packet type.

CL_REQUEST A CL broadcasts a CL_REQUEST packet to discover subordinate CLs. This packet is flooded through the network, but is dropped at all nodes that are not subordinates of the request CL (this is determined by comparing sender and node ID). For example, when the CL of Floor A sends out a CL_REQUEST, nodes on Floor B would drop the packet. As this packet is propagated through the network, nodes can set up routing tables to the originating CL. Therefore, a CL_REQUEST from a CL at hierarchical level n, will result in filling all the routing tables for level n at all nodes who recognize the sender as their CL.

CL_REPLY Every node that receives a CL_REQUEST packet from a parent CL answers with a CL_REPLY packet. This indicates that the replier is a potential candidate to be a CL at the sublevel. Reply packets use the routes to the CL established through the CL_REQUEST packet. The parent CL then chooses as an unique CL for each direct sublevel the quickest replier among all candidates with the lowest hop count. Other metrics such as signal strength for single hop candidates are also plausible. Intermediate nodes forwarding CL_REPLY packets will increment the hop count field. They can also drop packets from same-level nodes, if they have already sent a reply packet that is superior according to the well-known selection metric. As an example, a node in Room B can drop reply packets from other nodes in Room B, if they have a higher hop count than a previously forwarded packet.

CL_CONFIRM Finally, the parent CL sends a CL_CONFIRM packet to each chosen CL, which is flooded until it reaches the new CL. A confirmed CL then restarts the process by sending out its own CL_REQUEST to the next lower level of hierarchies.

4.3 Data Cloaking

Nodes employ two basic techniques to increase anonymity: Provide less spatial accuracy and perturb the count of subjects in the covered area. In our hierarchical organization, less spatial accuracy can be achieved by omitting a range of the less significant bits of the sender node ID (ID blurring); thus, the two approaches are:

1. Cloak ID, provide precise data
2. Cloak data, provide precise ID

The data cloaking algorithm combines both approaches. Each node stores the desired anonymity level k, which is preconfigured. If the number of subjects meets or exceeds k the algorithm cloaks data and provides a precise node ID (which describes the area); otherwise, it provides precise data with a cloaked ID.

Data cloaking is achieved through smart rounding. We define the smart rounding function as follows:

$$y = \begin{cases} x & \text{if } x \bmod k = 0 \\ round_k(x \quad (0.5 * r)) & \text{otherwise} \end{cases}$$

where $round_k$ rounds to the nearest multiple of k and r is a random variable that contains 0 or 1 with equal probability. At higher event counts, smart rounding will allow for more precision of data location, rather than actual data values. Smart Rounding would occur only once if no aggregation occurred with other data packets and then be passed directly up to the highest level.

Having the ID blurred at lower subject counts will allow aggregation of these small numbers to occur at higher hierarchical levels. Eventually the blurred ID will get to a hierarchical level where it can be aggregated and exceeds k. Otherwise, it will get passed up to the highest hierarchical level with that highest level's ID.

In order to defend against traffic analysis attacks, the network will follow a near constant rate of data traffic. Already, nodes are sending events to CLs at a constant rate. Of course, lack of traffic could also possibly give away information. In the office usage space example, lack of traffic would relate to no movement meaning that no one is in the room, floor, or even building. This is information we do not want an attacker to gain. Therefore all nodes are required to send at least one packet per data gathering interval (with an event count of zero). Even if a node has seen no events, it still must send a packet. CLs are also required to send at least one packet.

We require *at least* one packet, because CLs may have to send multiple packets due to the size of the data incoming versus the buffer size available on the node. By allowing more than one packet to be sent, our design is also much more scalable to larger networks, where data incoming to a node may completely overwhelm the resources available in a node.

5 Preliminary Conclusions

We have outlined a potential solution to the challenge of integrating privacy-enhancing mechanisms into sensor systems. This approach promises to strengthen user privacy protection compared to solutions at the database level because it *prevents collection* of privacy-sensitive data. From our ongoing work, we draw the following experiences and preliminary conclusions:

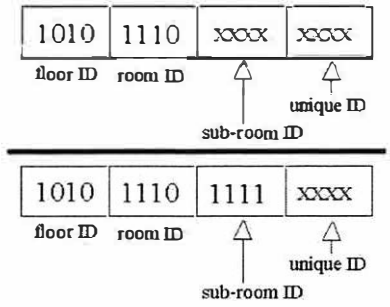


Figure 2: From the lower ID, a receiver would only know an event occurred at the sub-room level. The upper ID shows even more blurring to the room level

- Designing privacy protection into sensor systems seems feasible albeit the current design suffers from a substantial communication overhead to defend against traffic analysis. This is especially concerning, if sensors have a very restricted energy budget.
- Privacy concerns influence system design especially in the area of networking protocols.
- Needed is a formal, likely probabilistic, model for location anonymity that captures the notion of a continuous stream of data. This would enable a better evaluation of the privacy protection afforded by such systems.
- Finally, a better understanding of the location data accuracy requirements for different classes of applications would enable an analysis of the level of anonymity that can be sustained for such applications.

References

[1] CNN. Police: Gps device used to stalk woman. `http://www.cnn.com/2002/TECH/ptech/12/31/gps.stalk.ap/index.html`, December 31 2002.

[2] Marc Langheinrich. A privacy awareness system for ubiquitous computing environments. In *4th International Conference on Ubiquitous Computing*, 2002.

[3] Einar Snekkenes. Concepts for personal location privacy policies. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 48–57. ACM Press, 2001.

[4] Sastry Duri, Marco Gruteser, Xuan Liu, Paul Moskowitz, Ronald Perez, Moninder Singh, and Jung-Mu Tang. Framework for security and privacy in automotive telematics. In *2nd ACM International Worksphop on Mobile Commerce*, 2002.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 439–450. ACM Press, May 2000.

[6] Pierangela Samarati. Protecting Respondents' Identities in Microdata Release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6), 2001.

[7] Latanya Sweeney. Achieving k-Anonymity Privacy Protection Using Generalization and Suppression. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):571–588, 2002.

[8] Mike Spreitzer and Marvin Theimer. Providing Location Information in a Ubiquitous Computing Environment. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 270–283, 1993.

[9] Ginger Myles, Adrian Friday, and Nigel Davies. Preserving Privacy in Environments with Location-Based Applications. *IEEE Pervasive Computing*, 2(1):56–64, 2003.

[10] Urs Hengartner and Peter Steenkiste. Protecting Access to People Location Information. In *Proceedings of First International Conference on Security in Pervasive Computing (to appear)*, LNCS. Springer, Mar 2003.

[11] Marco Gruteser and Dirk Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (to appear)*, May 2003.

[12] Alastair R. Beresford and Frank Stajano. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[13] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The Cricket Location-Support System. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 32–43. ACM Press, 2000.

[14] Roy Want, Andy Hopper, Veronica Falco, and Jonathan Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

[15] Andy Ward, Alan Jones, and Andy Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4(5):42–47, Oct 1997.

[16] IBM Research Exploratory Computer Vision Group. Footprint: Infrared person tracking. `http://www.research.ibm.com/ecvg/misc/footprint.html`.

[17] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM Press, 2000.

[18] Hector Abrach, Jim Carlson, Hui Dai, Jeff Rose, Anmol Sheth, Brian Shucker, and Richard Han. MANTIS: System Support For MultimodAI NeTworks of In-situ Sensors. Technical Report CU-CS-950-03, University of Colorado, Department of Computer Science, April 2003.

[19] P. Samarati and L. Sweeney. Protecting Privacy when Disclosing Information: k-Anonymity and its Enforcement through Generalization and Suppression. Technical Report SRI-CSL-98-04, Computer Science Laboratory, SRI International, 1998.

[20] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, pages 189–199. ACM Press, 2001.

FAB: enterprise storage systems on a shoestring

Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence and Alistair Veitch
Storage Systems Department, Hewlett-Packard Laboratories, Palo Alto, CA
{frolund,arif,ysaito,suspence,aveitch}@hpl.hp.com

Abstract—A Federated Array of Bricks (FAB) is a logical disk system that provides the reliability and performance of enterprise-class disk arrays, at a fraction of the cost and with better scalability. The unit of deployment in FAB is a *brick*, a small rack-mounted storage appliance built from commodity components including disks, a CPU, NVRAM, and network cards. Bricks federate themselves in a completely decentralized manner to provide users with a set of logical volumes. This paper motivates FAB and introduces our data replication algorithm based on majority-voting. We argue that majority voting is practical for ultra-reliable, high-throughput storage systems like FAB, and present several techniques that improve both the performance and space overhead of our protocol.

1 Introduction

Disk arrays are today's standard solution for enterprise-class storage systems. The key requirement that separates disk arrays from consumer-class storage systems is their absolute reliability: a disk array must never lose data or stop serving data, under any circumstances short of complete disaster. To fulfill this requirement, disk arrays are constructed from customized, very reliable, hot swappable hardware components. Designing and building the hardware components is time-consuming and expensive, and this, coupled with relatively low manufacturing volumes, is a major factor in the high price of storage systems—high-end arrays retail for many millions of dollars.

Another cost factor, and problem for customers, is the lack of scalability of a single system. There is a high up-front cost for even a minimally configured array, and a single system is limited in both capacity and throughput. Many customers exceed these limits, resulting in poor performance or a requirement to purchase multiple systems, both of which increase management costs. The lack of scalability forces manufacturers to build multiple products, or even entire product lines, each targetted at different system scales. For example, Hewlett-Packard sells three different array lines, each of which effectively multiplies the engineering effort required — for hardware and firmware development, for integration and testing, etc, costs which are reflected in the price paid for each system.

A Federated Array of Bricks (FAB) is a low-cost alternative to disk arrays, and is designed to be scalable from very small to very large systems. FAB achieves this by composing together storage *bricks*, where each brick is a small rack-mounted storage appliance built from commodity components including disks, a CPU, NVRAM, and network cards. FAB systems cost much less than disk arrays to manufacture and develop, due to the economies of scale inherent in volume production, and because FAB can replace entire array product lines (amortizing development costs). Because of these factors, we anticipate that a FAB system can be built for far less than the equivalent high-end system. FAB provides comparable reliability, achieved through replication: we store the same disk block on multiple bricks, and we create redundant paths between all components of the system. FAB performance scales by completely distributing all functionality (no centralized bottlenecks) across the set of available bricks.

1.1 FAB: challenges and overview of solutions

We have identified the following key challenges to building a large, completely distributed storage system:

Failure tolerance: FAB is built from commodity hardware, which has empirically been found to be less reliable than the hardware used for enterprise systems [3, 2]. Every component—disks, bricks, networks can and will fail. FAB must seamlessly handle failures without data loss or delays in response to client requests.

Single-copy consistency: We must ensure that a replicated disk block logically looks like a single, highly available block to the client, even though there is no centralized software that oversees the I/O activities of the entire system.

Asynchronous coordination: We cannot rely on disks and operating systems to always act in a timely manner—e.g., an I/O request to a busy disk is known to sometimes take more than 5 seconds to complete (i.e., stuttering failures). Thus, we must coordinate replicas without any assumptions about their speed or network connectivity.

Hardware heterogeneity: As disk and CPU technologies evolve over time, the design of bricks will also evolve. We must let customers deploy different types of bricks

incrementally as their demand grows. FAB must assign resources to volumes in a way that maximizes overall performance and reliability.

FAB uses a quorum-based replication scheme, as described in Section 3, to address the first three challenges. In FAB, a "read" or "write" request completes when a majority of the replicas are functional. We recover from failures lazily, repairing the replicas during the next "read" request without any lock-step synchronization. Our protocol does not rely on failure detection—it just ignores dysfunctional components. It tolerates non-Byzantine failures, including network partitioning and stuttering failures. FAB uses dynamic load balancing and background reconfiguration, as discussed further in Section 4, to address the fourth challenge.

1.2 Related work

The idea of building a distributed logical disk from a decentralized collection of smaller components was pioneered by DataMesh [13] and Petal [10]. FAB extends Petal's ideas with better replication, volume layout, and load balancing algorithms. IBM's IceCube [8] builds innovative hardware for a FAB-like composable storage system, but we do not know their software structure yet.

Many high-throughput data systems, including Petal and most relational database systems, use some form of primary-backup replication. They fail to solve the challenges outlined in the previous section. In these systems, a failure of the primary renders its data unavailable until a new primary is elected. The actual fail-over time in these systems can be quite substantial. Having too short a fail-over time increases the chances of electing a new primary before the old primary has actually failed, consequences of which range from severe performance degradation (as in some group membership protocols) to outright data corruption (as in a naïve timeout-based failure detection scheme). Thus, in practice, these systems must conservatively choose a large fail-over period, often longer than 30 seconds, which actually causes the clients to time out.

The goal of [1] is to allow clients of a storage-area network to directly execute RAID encodings across distributed storage devices. This algorithm relies on the ability of clients to accurately detect the failure of storage devices. Moreover, the algorithm in [1] can result in data loss when certain combinations of client and device failures occur. In contrast, our algorithm can tolerate the simultaneous crash of all bricks, and it can make progress whenever a majority recover and are able to communicate.

Numerous replication protocols use majority voting. The protocol by Thomas [12] is similar to ours in that it uses timestamps to order "write" requests against a majority of replicas. However, Thomas' protocol only guarantees convergence, which is too weak for a distributed logical disk. FAB guarantees linearizability [7]. Several state-machine replication algorithms, such as Paxos [9], use majority voting to achieve a total order for requests. Our replication protocol exploits the semantics of read and write operations to achieve the same thing in fewer rounds, using less space. Messaging-based atomic register algorithms [4, 11] resemble our algorithm the most; they use majority voting and exploit read and write operation semantics. These algorithms, however, require more rounds (especially in the common case) than ours and lack support for process recovery.

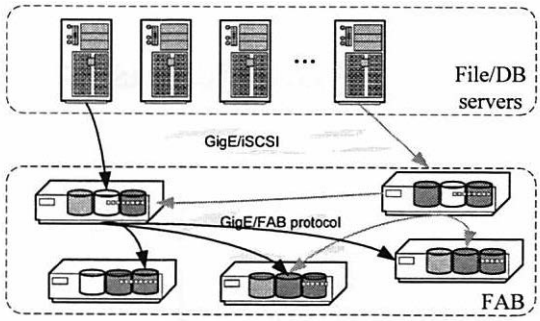


Figure 1: A typical FAB structure. Client computers connect to the FAB bricks using standard protocols. Clients can issue requests to any brick to access any logical volume. The bricks communicate among themselves using our replication protocol.

2 Structure of FAB

Figure 1 shows the structure of FAB. Client systems connect to FAB bricks using standard protocols such as Fibre Channel or iSCSI. Bricks are connected to each other using standard local-area networks, such as 1 Gbps Ethernet. FAB presents the clients with a number of logical volumes, each of which may be accessed transparently as if it were a single disk. Since FAB is a decentralized system without a central management node, a client can ask any brick to create, resize, or access a logical volume. Bricks use a custom protocol, described in the next section, to coordinate among themselves and provide a consistent view of volumes. A single FAB system is anticipated to contain up to 5000 bricks with a logical capacity of 2 petabytes.

FAB internally splits each logical volume into fixed-size *segments*. Each segment contains a number of *blocks*, the minimum unit of access. The segment size and block size default to 8GB and 1KB, respectively. A number of segments are gathered into *groups*. Each group is replicated over several bricks (three by default: see below), chosen randomly out of the set of bricks with available space. The use of segments and groups enables efficient metadata management; segments are used in layout management and groups for replication and availability.

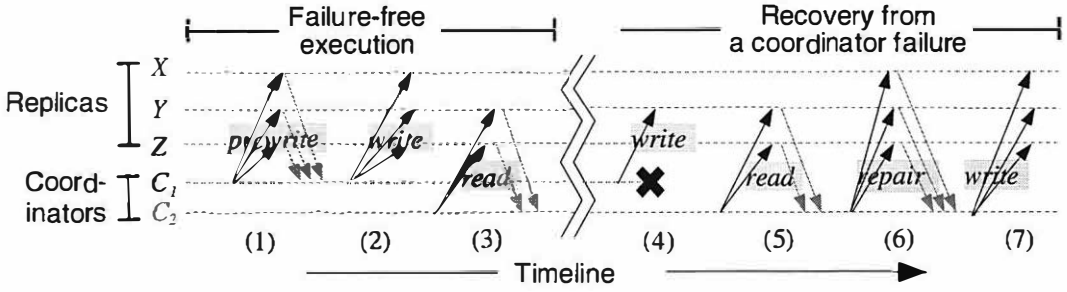


Figure 2: In this example, a disk block is replicated on three bricks, X, Y, and Z. Two coordinators, C_1 and C_2, issue requests to the replicas. The first scenario (steps 1 to 3) shows a failure-free execution. Brick C_1 writes to the block in two rounds. In the *prewrite* round, the replicas update their *logTs* to indicate a new ongoing update and promise not to accept any request older than this request. In the *write* round, the replicas actually write the new value to their disks and sets *ts* to indicate that the update is complete. In step (3), node C_2 reads blocks from a majority of $\{X,Y,Z\}$, discovers that the block contents are consistent and finishes (in practice, C_2 reads the block value from only one replica; see Section 3.1.) The second scenario (steps 4 to 7) shows why the *prewrite* round is needed. Here, C_1 tries to write, but crashes just after sending the *prewrite* to only Y. Later, while trying to read, C_2 discovers that $ts \neq logTs$ on Y; i.e., the replicas are inconsistent because of an incomplete write request. C_2 runs the *repair* round in step (6) on a majority of the replicas to discover the newest value, and writes it back to (at least) the majority of the nodes in step (7), so that future requests will never read older values.

Each brick internally runs three software modules: the *coordinator* module that receives client requests and coordinates disk read or write requests on behalf of clients, the *block-management* module that actually reads and writes disk blocks, and the *configuration-management* module that oversees administrative changes. The next section describes the interaction between the coordinator and block-management modules. The configuration-management module uses the Paxos distributed consensus algorithm [9] to replicate configuration information—e.g., the set of bricks that exist in the system, and the name and layout of logical volumes---on all bricks. We plan to investigate a more efficient configuration management scheme in the future.

The choice of a redundancy scheme to provide high reliability at an acceptable cost is an important consideration for FAB. We have considered erasure coding and replication. Erasure coding has high space efficiency and high reliability, but poor performance when bricks fail (reads access multiple bricks and writes require a read before writing). Instead, FAB uses replication of data across multiple bricks to overcome brick failures. We compared the reliability of 2-way and 3-way replication schemes by computing their mean time to data loss (MTTDL) and mean unavailability using component failure rates from Asami [3] and by assuming that data is lost when all bricks holding replicas of any segment fail. We consider a FAB system of 256 bricks; each brick uses RAID-5 across 12 SATA disks holding replicas of 128 data segments. Each segment group contains 32 segments. For 2-way replication, we estimate a MTTDL of 267 years and a mean unavailability of 0.02% (1.8 hours/year) which is inadequate for critical applications. For 3-way replication, the MTTDL is 1.3 million years and the mean unavailability is 3×10^{-6}% (1 second/year), which is acceptable. Based on these considerations, we chose 3-way replication as the default policy, but we also allow administrators to choose other replication factors.

Creating three replicas for each logical block sounds expensive, but is only 33% more capacity intensive than RAID-10; and FAB systems are build from cheaper components than existing high-end disk arrays, which reduces the cost substantially. In Section 5, we discuss our plan to use "witness" replicas to reduce the effective storage consumption, while maintaining an acceptable level of reliability.

3 The FAB replica-management protocol

This section describes FAB's approach to replica consistency. We first introduce the basic protocol for maintaining replicated blocks. Later sections describe performance optimizations and extensions to optimize the system's performance and memory consumption.

In FAB, an I/O request to logical blocks is handled by the coordinator module of any brick (from a clients view, every brick can act as a disk array controller). FAB runs a variation of a timestamp-based majority-voting protocol [12]. The full details of the protocol and a correctness proof are presented in [5].

Figure 2 shows an example. The task of the coordinator is straightforward in theory: when writing, it generates a new timestamp and writes the value and timestamp to the majority of replicas; when reading, it reads from the majority and returns the value with the newest timestamp.

The failure of the coordinator itself, however, causes a problem, because it may leave a new value on a sub-

Workload	*date*	*length*	*volume size*	*#writes*	*#reads*	*data written*	*data read*	*unique data written*
Cello	9/2002	1 day	1.4 TB	5,250,126	6,766,002	67.4 GB	160GB	27.1 GB
SAP	1/2002	15 min	5 TB	150,339	4,835,793	1.75 GB	55.4GB	1.36 GB
OpenMail	10/1999	1 hr	7 TB	931,979	355,962	61.3 GB	2.47 GB	1.64GB

Table 1: Workload characteristics. *Date* shows when the trace was collected. *Unique data written* is the amount of data written once overlapping writes are removed.

majority of the replicas. A logical disk system must ensure *linearizability* [7]—roughly speaking, all clients must see a single global ordering of (either successful or failed) read and write requests for each logical block, even when these requests are coordinated by different bricks. Thus, after a coordinator failure, future "read" requests on the same block must all return the old block value or all return the value attempted by the failed coordinator (unless the block is overwritten by a newer "write" request). Previous approaches, such as those using two-phase commits [6], cannot ensure a quick fail-over. FAB takes an alternative approach, performing recovery in a lazy manner when a client tries to read the block for the first time after the failure.

To detect the partial writes that result from failures, each replica of a logical block keeps two timestamps: the *ts* is the timestamp of the value currently stored, whereas the *logTs* is the timestamp of the newest ongoing "write" request. As illustrated in Figure 2, a "write" request runs in two phases, using the timestamps to ensure linearizability. A "read" request usually runs in one phase, but takes three phases when timestamp state indicates the past failure of a (different) coordinator: the value with the highest timestamp is stored in a majority with a timestamp greater than that of any previous writes, including any partial writes.

We assume that clients have multi-path capability, so the failure of a coordinator does not stop them issuing requests to FAB. Since FAB requires no change to clients, the clients' own software and the standard protocol used between clients and FAB dictate client reaction to coordinator failure.

3.1 Improving the efficiency of majority voting

Majority voting has been proposed as a simple yet robust replication method for quite a while [6, 12], but no system has used it in a high-throughput environment. The oft-encited reason is that it is inefficient, because "read" requests must contact multiple remote replicas [14]. This reason, however, does not apply to FAB for the two reasons.

First, we apply an "optimistic read" technique for the common case scenario of reading from a (logical) block that is already consistent. Here, the coordinator reads the actual block contents from one idle replica and reads only timestamps from others in the quorum. This technique, in effect, reduces the number of disk accesses to one per "read" request, as the vast majority of timestamps will be cached in main memory for the reasons described in the next section.

Secondly, FAB is a naturally disk-I/O-bound system; the CPU and network spend most of the time waiting for disk I/Os to complete. The overhead of extra timestamp processing does not slow the system down.

3.2 Reducing the overhead of timestamp management

One challenge that FAB faces is the timestamp management overhead: for every 1 TB of data, with a 12 byte timestamp recorded for every 1KB block, 12 GB of space could potentially be required to store timestamps. This information must be kept persistently, yet this amount of NVRAM is infeasible. We employ several techniques to reduce the overhead of timestamp management substantially.

First, we observe that timestamps are used only to disambiguate concurrent updates and to "repair" the results of previous failures. Thus, in the case where all replicas of a logical block are functional, timestamps can be discarded once all the replicas have acknowledged an update. Replies to the client are made as soon as a majority of the replicas have acknowledged an update. The coordinator, in the background, runs a third phase to write processing in which it lets replicas remove their timestamps once all have replied. In the normal case, a brick needs to keep timestamps only for blocks that are actively updated; these timestamps can easily be kept in NVRAM. After one of the replicas fails, other replicas must keep timestamps for blocks that are updated, until the replica recovers or a reconfiguration starts. However, as we show below, it is extremely unlikely that the number of these timestamps exceeds what a brick can store in memory.

A second optimization can be made by observing that a single "write" request almost always updates multiple blocks, and that each of the blocks affected will have the same timestamp. We thus keep timestamp information on ranges of blocks, rather than per-block.

To investigate the impact these optimizations would have, and to determine the actual amount of memory needed, we have analyzed several real-world I/O traces, summarized in Table 1.

Workload	*raw*	*written*	*multiple*
Cello	16.8 GB	26.4 MB	228 MB
SAP	60GB	128 MB	10.3 MB
OpenMail	84 GB	38.4 MB	4.00MB

Table 2: Timestamp memory requirements. *Raw* is the amount of memory required with a timestamp for every 1KB block, *written* is the amount required if timestamps are only kept for blocks written during the trace, *multiple* is the amount if a timestamp covers multiple blocks. For sparse structures, we assume a doubling over the raw timestamp size to allow for the data structure overhead. All numbers except for *raw* (which does not vary over time) have been normalized to be per-hour, i.e., the amount of memory required for every hour of operation while a replica is unavailable. This normalization results in pessimistic estimates—as significant spatial locality is shown over time, the rate of timestamp generation decreases when longer time periods are observed. For a longer, 3 day "Cello" trace, the rate of timestamp generation is half of that shown by the 3rd day.

Cello: A file system managed by an 8 processor HP9000 N4000 for 20~30 researchers, with 16 GB of RAM, and an HP XP512 disk array.

SAP: A SAP ISUCCS 4.5B and Oracle supporting 3000 users and several background batch jobs running on an HP V2500 with an HP XP512 disk array.

OpenMail: An HP9000 K580 server with 6 CPUs, 3.75 GB of RAM, and an EMC 3700 Symmetrix disk array. Approximately 2000 users access their email during the course of the trace.

Table 2 shows the amount of memory required for timestamp information under various circumstances. These results show that even if a system is down for several days, even a relatively modest amount of memory will be sufficient to store all the timestamps needed.

4 Data layout and load balancing

As discussed in Section 2, FAB chooses the set of replicas for each segment randomly. The randomized layout has many advantages over a deterministic mapping such as chained declustering [10]: the load is uniformly distributed over the bricks; when bricks leave FAB, reassigning the segment replicas on these bricks to other bricks is straightforward; similarly, when new bricks join FAB, reassigning segment replicas from heavily loaded bricks to the new bricks is easy to do; and non-homogeneous bricks with different capacities and performance characteristics can be handled. Moreover, by using reasonably large segments (say, 8GB) the size of the layout table can be kept small.

The FAB replica-management protocol permits actual data reads to be made from any replica; the other replicas only provide timestamp information. By having coordinators perform the data read from the least loaded replica, load can be shifted away from a heavily loaded brick to its neighbors (bricks holding replicas of the segments on the heavily loaded one), which can further shift read load to their neighbors, and so on, until the entire FAB shares the load. This mechanism makes it possible to accommodate bricks with heterogeneous performance. It also makes FAB highly resilient to load imbalance due to brick failures, since the read load from failed bricks is automatically spread over the entire FAB. More intelligent load balancing may be considered in future work.

5 Current status and future work

We have implemented a prototype and are studying its behavior under various situations, including failures and overloads.

We identify two major areas of future work. One is dynamic volume reconfiguration after failures or to improve performance. The requirement remains the same: linearizability, asynchronous coordination, and no service stoppage during reconfiguration. We plan to adapt the technique described in [11], by superimposing a new quorum configuration using Paxos, transferring contents to new bricks, and garbage collecting old quorum configurations in the background.

The other is reducing the storage overhead of quorum-based replication using *witnesses* and *witness promotion*. We adapt the timestamp-discarding scheme introduced in Section 3.2 to create "witness" replicas that only keep timestamps, but no actual block values (at least in the long term). By replicating a logical segment on only $f+1$ normal replicas and f additional witnesses, the segment can tolerate f failures with little space overhead. A witness participates in the block-I/O protocol exactly like other replicas—it actually stores block contents in a scratch disk area. When it receives a "discard timestamps" request for a block, however, it recycles the disk area. In the common case where all replicas are functional, a witness only consumes disk blocks for ongoing updates. When a witness accumulates too many disk blocks for outstanding updates after a failure of another replica, it eventually promotes itself into a full-fledged replica using the aforementioned reconfiguration algorithm.

In the longer term, we also need to evaluate the full cost of ownership of a FAB system, including maintenance in the face of the higher expected rate of component failures, power and cooling expenses, and to address the potential engineering problems of a FAB system at the scale of 5000 bricks.

6 Acknowledgements

We would like to thank other members of the HP Labs Storage Systems Department for feedback, particularly John Wilkes. Our shepherd, Jeff Chase, provided valuable comments that improved the paper.

References

[1] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan, April 2000.

[2] D. Anderson, J. Dykes, and E. Riedel. More than an interface—SCSI vs. ATA. In *USENIX Conf. on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[3] S. Asami. *Reducing the cost of system administration of a disk storage system built from commodity components.* PhD thesis, University of California, Berkeley, May 2000. Tech. Report. no. UCB-CSD-00-1100.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[5] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. Building storage registers from crash-recovery processes, May 2003. Tech report HPL–SSP–2003–14, available at http://www.hpl.hp.com/research/ssp/papers/.

[6] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th. Symposium on Operating Systems Principles*, 1979.

[7] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 12:463–492, 1990.

[8] IBM. IceCube: storage server for the Internet age. http://www.almaden.ibm.com/cs/storagesystems/IceCube/.

[9] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001. http://research.microsoft.-com/users/lamport/pubs/pubs.html.

[10] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th Int. Conf. on Architectural Support for Prog. Lang. and Op. Systems*, pages 84–92, Cambridge, MA, 1996.

[11] N. A. Lynch and A. A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *16th Int. Conf. on Dist. Computing (DISC)*, pages 173–190, Toulouse, France, October 2002.

[12] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys. (TODS)*, 4(2):180–209, June 1979.

[13] John Wilkes. Datamesh research project, phase 1. In *Proc. USENIX Workshop on File Systems*, pages 63–69, Ann Arbor, MI, May 1992.

[14] Avishai Wool. Quorum systems in replicated databases: science or fiction? *Bull. IEEE Technical Committee on Data Engineering*, 21(4), December 1998.

The Case for a Session State Storage Layer

Benjamin C. Ling and Armando Fox
Stanford University
{bling, fox} @ CS.Stanford.edu

Abstract

This paper motivates the need for a session state storage layer. Session state is used in a large class of applications. Existing session state storage solutions often rely on ad-hoc choices such as databases or file systems, and exhibit various drawbacks such as poor failure/recovery behavior or poor performance, often because the solutions are *too general*. We present a design for a simple, fast, scalable, and fault-tolerant session state store that we believe accurately addresses the needs of session state retrieval, while avoiding the drawbacks of existing solutions.

Introduction

The concept of a user session is present in nearly all client-facing applications, including web-based applications. A user actively works for a period of time, called a *session*, until he signs out, or his session expires after a fixed interval. During the session, the application may produce temporary data relevant to the user's session, e.g. which step the user has completed in the application workflow. Upon session completion, the temporary state is no longer needed.

Session state is state whose lifetime is the duration of a user session and is relevant to a particular user; the duration of a user session is application specific. Examples of session state are user workflow state in enterprise software and user navigation in eCommerce.

Many architectures have modules that produce and use session state, including emerging industry standards such as J2EE[1]. Regardless of architecture, session state and its storage is a building block that is useful. Session state is a large class of state, as there are many different types of data that can be generated and used depending on the application at hand. In this paper, we focus on an interesting subset of session state, with distinct requirements and properties, which we describe in the upcoming sections.

In Section 1, we present an example of how session state is used. Next, we discuss the requirements and properties for the subset of session state that we address, and discuss the functionality that is necessary to provide a session state store, also highlighting what is *not* necessary. We then discuss current solutions and why they are inadequate. We propose a "middle-tier" storage layer to address session state. We discuss future work and related work, and conclude.

1. What is session state?

In Section 1, we describe a large subcategory of session state by describing how it is used, its properties, the requirements it imposes on its storage, as well as the functionality required to support it. Various different categories of session state exist. However, in the remainder of this paper, we will use the term "session state" to refer to the *subcategory* of session state which we describe below.

We use the example of a user working on a web-based marketing application to illustrate how session state is often used. The user is building a marketing campaign, specifying target customers and the offers they should receive.

A large class of applications, including J2EE-based and web apps in general, use the interaction model below:

- User submits a request (to add a targeted customer), request is routed to a stateless application server. This server is often referred to as the middle-tier.
- Application server retrieves the full session state for user (which includes the campaign data).
- Application server runs application logic (adds a customer to the targeted set of customers)
- Application server writes out entire session state
- Results are returned to the user's browser

Session state must be present on each interaction, since user context or workflow is stored in session state. If it is not, the user's workflow and context is lost, which is seen as an application failure to the end user, and is usually unacceptable from a product requirement standpoint. Session state, in this context, includes any temporary application state that is associated with a single user; the loss of this state is akin to losing a few hours work.

Typically, session state is on the order of 3K-200K [2]. Session state retrieval is also in the critical path of the control path – processing of the request cannot continue unless session state has been retrieved.

These requirements imply that session state solutions should have the following properties, besides traditional properties such as availability, scalability, performance:

- Session state retrieval should be fast, or negligible when compared to application processing
- Failures of the state store or any of its subcomponents should not result in data loss, otherwise users perceive an application failure

- Recovery of the state store and its failed subcomponents should be fast, for the same reason

Some important properties/qualities of the session state we focus on are listed below. Session state:

1. **Is not shared.** Each user reads his own state. Unlike state in its full generality, session state is accessed in a fixed pattern of alternating reads and writes: *Read 1* of session state for user U is followed by *Write 1*, which is followed by *Read 2*, followed by *Write 2*.
2. **Is semi-persistent.** Session state must be present for a fixed interval *T*, but can be deleted after *T* has elapsed.
3. **Is keyed to a particular user.** An advanced query mechanism to do arbitrary searches is not needed.
4. **Is updated on every interaction.** Session state such as user context in a web-based application is updated in its entirety on every interaction, as described earlier. A new copy of the state is written on every interaction.
5. **Does not need ACID [7] semantics.** Session state is transient, and state that requires transactions is not included in the class of session state we address.

Given these properties, the functionality necessary for a session state store can be greatly simplified (each point corresponds to an entry in the previous numbered list):

1. **No synchronization is needed.** Since the access pattern corresponds to an access of a single user making serial requests, no conflicting accesses exist, and hence race conditions on state access are avoided, which implies that locking is not needed.
2. **State stored by the repository need only be semi-persistent** – a temporal, lease-like [3] guarantee is sufficient, rather than the "durable" guarantee that is made in ACID [7].
3. **Single-key lookup API is sufficient.** Since state is keyed to a particular user and is only accessed by that user, a general query mechanism is not needed.
4. **Previous values of state keyed to a particular user may be discarded.**
5. **No need to support full ACID** – only atomic update is necessary; since each write writes out all of the user's session state, consistency is trivial and isolation is guaranteed. Durability is not needed.

Existing solutions and why they are inadequate

Currently, session state storage is done with one of the following mechanisms: Relational Database (DB), file system (FS), single-copy in-memory, replicated in-memory.

Frequently, enterprises use either the DB or FS to store session state, often reasoning, "I already have a DB and FS, why don't I just use one of them to store session state?" This simplifies management, since only one type of administrator is needed. However, there are several drawbacks to using either a DB or FS to handle session state, besides the costs of additional licenses and complexity of administration:

- **D1 Contention.** Unless a separate DB/FS is created for session state, requests for session state and requests for persistent objects contend for the same resources. Session state read/write requests are frequent, which can interfere with requests for persistent objects that are housed by the same physical resource.
- **D2 Failure and recovery is expensive.** If a crash occurs, recovery of the DB or FS may be slow, often on the order of minutes or even hours. Recovery time for a DB can be reduced if checkpointing is done frequently, but reduces performance under normal operation. There exist DB/FS solutions that have fast recovery, but these tend to be quite costly [9]. Even if recovery is on the order of seconds, in a large scale application, hundreds or thousands of users may see a failure if they attempt to contact the server at time of recovery.
- **D3 Session cleanup is painful.** After state is put into a DB or FS, some process has to come back and look at the data and expire it, or else the data continues growing without bound. Reclaiming expired sessions degrades performance of other requests to the DB or FS.
- **D4 Potential performance problems.** Reading/writing state objects to a DB/FS may sometimes incur a disk access in addition to a network roundtrip.

On the other hand, in-memory solutions (IMS) avoid several of the drawbacks of FS and DB, and are generally faster than FS/DB oriented solutions. In-memory solutions rely on affinity, and require a user to "stick" to a particular server that stores his copy of session state. A hardware load-balancer can guarantee affinity. However, the app-processing tier is no longer stateless; session state is being stored by the application server; it must serve the dual roles of application processing as well as providing state storage. Affinity is a key property for in-memory solutions to operate well – the main advantage of storing state in memory is to avoid a network roundtrip to the DB. If no affinity is present, then a server must incur a roundtrip to pass the request to the appropriate server. Affinity limits load balancing options, since load balancing must be done on the granularity of a user, rather than that of a request.

When only a single copy of a user's session state is stored on a corresponding application server, if a server crashes, state for some users is lost. The crash will be

manifested to users as an app failure, which is usually unacceptable.

A primary-secondary scheme is often used for a replicated solution. In BEA WebLogic™ [5], a J2EE application server, servers are given unique IDs and form a logical ring. A server S elects the server T that trails S in the logical ring to be its secondary. All users who are pinned to S as a primary share T as a secondary. A cookie is written out to the user's browser designating the primary and secondary.

During nonnal operation, all updates to session state are synchronously written, first at the primary, and then synchronously replicated at the secondary.

On failure of S, WebLogic™ does one of the following, depending on configuration.

1. A subsequent request destined for S is assigned to a random server U, because the load balancer recognizes the failure of S. U will copy the state information from the secondary T, and then rewrite the user's cookie, designating U as the new primary and T as the secondary.
2. A subsequent user request destined for S is assigned to the secondary T, because the software load balancer recognizes the failure of S. T will designate itself as the primary, replicate the session state information to its secondary V, and then rewrite the user's cookie, designating T as the new primary and V as the secondary.

There are several potential problems in this scheme (Note that some of the deficiencies of DB/FS solutions are shared by WebLogic™, as mentioned below):

D5 Performance is degraded on secondaries. D5 is related to D1. Instead of only providing application processing, secondary application servers face contention from session state updates.

D6 Recovery is more difficult (special case code for failure and recovery). The middle-tier is now stateful, which makes recovery more difficult. Special-case failure recovery code is necessary. In Case 1, a server A receiving a valid cookie stating that B as primary and C as secondary must realize that it must now become primary since B failed, and in Case 2, a secondary must realize that it should now become the primary. Special-case code makes the overall system harder to reason about, harder to maintain and less elegant.

D7 Poor failure/recovery performance for Case 2. Assuming equal load across all servers, upon failure of a primary A, the secondary B will have to serve double load – B must act as primary for all of A's requests as well as its own. Similar logic applies to B's secondary, which experiences twice the secondary load.

D8 Lack of separation of concerns. The application server now provides state storage, in addition to application logic processing. These two are very different functions, and a system administrator should be able to scale each separately.

D9 Performance coupling. If a secondary is overloaded, then users contacting a primary for that secondary will experience poor performance behavior as well. Because of the synchronous nature of updates from primary to secondary, if the secondary is overloaded, e.g. from faulty hardware or user load, the primary will have to wait for the secondary before returning to the user, even if the primary is under-loaded [4]. An industry expert has confirmed that this is indeed a drawback [6].

Note that *any* replication scheme requiring synchronous updates will *necessarily* exhibit performance coupling. That is, whenever a secondary is slow for any reason, any primary served by that secondary will block. In Case 1, after a failure of a single server, the entire cluster should be coupled, if we assume that the load balancer load balances correctly. To be specific, each request for S will be assigned to a random server U, and all of the nodes in the cluster will be performance coupled to the secondary T. This is particularly worrisome for large clusters, where node failures are more likely because of the number of nodes. Furthermore, application servers often use shared resources such as thread pools, and slowness in the secondary will hold resources in the primary for longer than necessary.

Proposed solution: A "middle-tier storage" layer

The support of industry in J2EE, together with the special qualities of session state and the failure of current solutions to address it properly, presents a viable opportunity to innovate and explore new state storage solutions. We present a solution that focuses on the following principles:

P1 Avoid special case recovery code. This addresses deficiencies D2 and D6. Avoiding special case recovery code reduces total cost of ownership, by facilitating easier administration and allowing programmers to reason about the system more easily, since the "special failure case" is not special, but rather the normal case [13].

P2 Design for separation of concerns. A system administrator should be able to scale the DB/FS separately from the session state store, and scale the session state store separately from the application processing tier. This addresses D1 D5, and D8.

P3 Session cleanup should be easy, and not require extra work. This addresses D3.

P4 Graceful degradation upon failure. Unnecessary performance degradation effects should not be seen, such as cache warming effects in DDS [4], and uneven load distribution in BEA. This addresses D7.

P5 Avoid performance coupling, as seen in DDS and which is present in the in-memory replication scheme. This addresses D9.

In addition, the system should be able to tolerate N simultaneous faults, where N is configurable by the system administrator. Unless N simultaneous faults occur, the system should continue operating correctly.

We assume a physically secure and well-administered cluster, along with a commercially-available high throughput, low latency redundant system area network (SAN) that can achieve high throughput with extremely low latency. High redundancy in the SAN enables us to assume that the probability of a network partition is arbitrarily small, and we need not consider network partitions. An uninterruptible power supply reduces the probability of a system-wide simultaneous hardware outage. The middle-tier storage has two components: bricks and stubs.

A brick stores session state objects by using a hash table. Each brick sends out periodic beacons to indicate that it is alive.

The stub is used by applications to read and write session state. The stub interfaces with the bricks to store and retrieve session state. Each stub also keeps track of which bricks are currently alive.

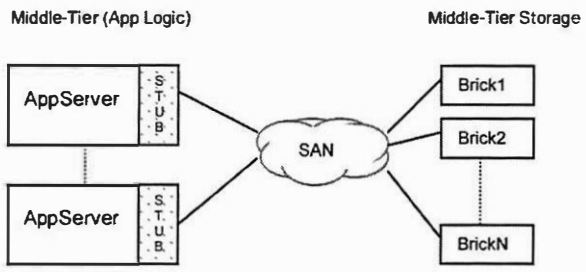


Figure 1

The write interface exported by the stub to the application is `Write(HashKey H, Object v, Expiry E)` and returns a cookie as the result of a successful write, or throws an exception if the write fails. The returned cookie should be stored on the client. The read interface is `Read(Cookie C)` and returns the last written value for hash key H, or throws an exception if the read fails. If a read/write returns to the application, then it means the operation was successful. On a read, we guarantee that the returned value is the most recently written value (recall that the type of session state we are dealing with is accessed serially by a single user).

The stub propagates write and read requests to the bricks. Before we describe the algorithm describing the stub-to-brick interface, let us define a few variables.

Call W the write group size. A stub will *attempt* to write to W of the bricks, and read from R bricks. Define WQ as the write quota, which is the minimum number of bricks that must return "success" to the stub before the stub returns to the calling application. We use the term quota to avoid confusion with the term quorum; quorums are discussed in section on related work. $WQ - 1$ is the number of **simultaneous brick failures** that the system can tolerate before losing data. Note that $1 \leq WQ \leq W$, $1 \leq R$, and $R \leq W$. Lastly, call t the timeout interval, an amount of time that the stub waits for bricks to reply to its requests.

The stub handles a write by doing the following:

1. Calculate checksum for object and expiration time.
2. Create a list of bricks L, initially the empty set.
3. Choose W **random** bricks, and issue the write of {object, checksum, expiry} to each brick.
4. Wait for WQ of the bricks to return with success messages, or until t elapsed. As each brick replies, add its identifier to the set L.
5. If t has elapsed and the size of L is less than WQ, repeat step 3. Otherwise, continue.
6. Create a cookie consisting of H, the identifiers of the WQ bricks that acknowledged the write, and the expiry, and calculate a checksum for the cookie.
7. Return the cookie to the caller.

The stub propagates the read to the bricks:

1. Verify the checksum on the cookie.
2. Issue the read to R random bricks chosen from the list of WQ bricks contained in the cookie.
3. Wait for 1 of the bricks to return, or until t elapses.
4. If the timeout has elapsed and no response has been returned, repeat step 2. Otherwise, continue.
5. Verify checksum and expiration. If checksum is invalid, repeat step 2. Otherwise continue.
6. Return the object to the caller.

The set of bricks that have the most recent write for key X changes on each write. While omitted in the algorithm for simplicity, a second-level timeout can be added for the case when writes/reads to the stubs continually time out; an exception can be thrown when this occurs.

For garbage collection of bricks, we use a method seen in generational garbage collectors [10]. Earlier we described each brick as having one hash table, for simplicity. In reality, it has a set of hash tables; each hash table has an expiration. A brick handles writes by putting state into the table with the closest expiration time after the state's expiration time. For a read, the stub also sends the key's expiration time, so the brick knows which table to look in. When a table's expiration has elapsed, it is discarded, and a new one is added in its place with a new expiration.

What happens on failure?

If a node cannot communicate with another, we assume it is because the other node has stopped executing. As discussed earlier, we assume that a network partition is not possible. We assume that components are fail-stop.

On failure of a client, the user perceives the session as lost, i.e. if the OS crashes, a user does not expect to be able to resume his interaction with a web application.

On failure of an application server, a simple restart of the server is sufficient since it is stateless. The stub on the server detects existing bricks from the beacons and can reconstruct the table of bricks that are alive. The stub can immediately begin handling read and write requests.

On failure of a brick, a simple restart of the brick is necessary. All state on that brick is lost; however, the state is replicated on $(WQ - 1)$ other bricks, and so no data is lost. Furthermore, on a subsequent write of that data, WQ copies are made, and the system can once again tolerate $(WQ - 1)$ faults without losing data.

An elegant side effect of having simple recovery is that clients, servers, and bricks can be added to a production system to increase capacity. For example, adding an extra brick to an already existing system is easy. Initially, the new brick will not service any read requests since it will not be in the read group for any requests. However, it will be included in new write groups because when the stub detects that a brick is alive, the brick is a candidate for a write. Over time, the new brick will receive an equal load of read/write traffic as the existing bricks.

When the system is under high load, and latency exceeds t, new requests will be generated while old ones have not yet been serviced, potentially increasing the load even more. To address this, bricks can discard a request if t has elapsed by the time the brick begins processing it. Secondly, we insert a random exponentially delay between each retry.

Middle-tier storage vs. design principles

We believe the design of this middle-tier storage system achieves the design principles outlined, and avoids the drawbacks of previous solutions while achieving good performance and exhibiting good failure/recovery behavior.

P1 Avoid special case recovery code. The system as described has no special case recovery code.

P2 Allow for separation of concerns. A middle-tier storage layer can be scaled separately from persistent storage usage and separately from application processing.

P3 Session cleanup is easy. Bricks can easily expire session by removing them from the hash. No extra process to scrub old data is needed.

P4 Node failure results in graceful degradation. Since multiple copies are available for any given write, a single node failure does not affect correctness, only the capacity of the system. Furthermore, multiple node failures do not affect correctness as long as the number of simultaneous failures is less than WQ.

P5 Performance coupling is avoided by two strategies: *Change the brick replica group on each write.* In schemes where a key is mapped to a fixed set of replicas (i.e. Key X is always mapped to replicas A, B, and C), performance for key X is limited by the slowest in the replica group. This implies that if the entire cluster is functioning correctly, with the exception of a single node, all requests served by the faulty replica group will experience poor performance. It is important to note that *any scheme requiring synchronous replies from a fixed set of nodes will experience negative performance coupling* – namely, performance is limited by the slowest in the group.

Issue more writes than necessary and wait for a few of them to return. By issuing a write to more replicas than are required, we avoid performance coupling caused by faulty nodes. Say bricks A, B, C compose the write group for a given write of key X, and B is faulty or overloaded. If WQ is set to 2, the faulty node does not hamper performance of the higher-level request. This is especially important since the replica group for key X changes on each write – if we do not issue the write to more bricks than necessary, in the presence of a faulty brick, eventually all keys will experience poor performance at one point or another.

The ratio of the tunable parameters WQ to W allows the administrator to determine the performance/cost tradeoff.

Interesting properties of solution:

An interesting property is a negative feedback loop involving the stubs and bricks. For example, let W be 3, WQ be 2, and the write group be A, B, C. If B is faulty or overloaded, A and C will reply first. A subsequent read will not involve B. In general, B will fail to reply to most write requests in time. By monitoring the number of operations handled by a brick, an administrator can detect and replace faulty nodes.

If B is temporarily overloaded, it can start shedding writes, to bring load back to a reasonable level. Furthermore, since subsequent reads for the state will not be addressed to B, load is reduced. This may result in writes failing under high load, but it is often better to shed load early, as evidenced in Internet workloads.

Related Work:

A similar mechanism is used in quorum-based systems [11, 12]. In quorum systems, writes must be propagated to W of the nodes in a replica group, and reads must be successful on R of the nodes, where $R + W > N$, the total number of nodes in a replica group. A faulty node will often cause reads to be slow, writes to be slow, or possibly both. Our solution obviates the need for a quorum system, since the cookie contains the references to up-to-date copies of the data; quorum systems are used to compare versions of the data to determine which copy is the current copy.

DDS [4] is very similar to the proposed middle-tier storage layer. However, one observed effect in DDS is performance coupling – a given key has a fixed replica group, and all nodes in the replica group must synchronously commit before a write completes. DDS also guarantees persistence, which is unnecessary for session state. Recovery behavior also exhibits negative cache

warming effects; when a DDS brick is added to the system, performance of the cluster first drops (because of cache warming) before it increases. This effect is not present in our work, since recovered/new bricks do not serve any read requests.

From distributed database research, Directory-Oriented Available Copies [15] utilizes a directory that must be consulted to determine what replicas store valid copies of an object. This involves a separate roundtrip, and the directory becomes a bottleneck. In our work, we distribute the directory by sending the directory entries to the browser, leveraging the fact that for a given key, there is a single reader/writer.

We share many of the same motivations as Berkeley DB [14], which stressed the importance of fast-restart and treating failure as a normal operating condition, and recognized that the full generality of databases is sometimes unneeded.

The need for graceful degradation upon failure was recognized in Petal [16]. However, Petal uses chained declustered; nodes are logically chained, and upon failure of a node, the node's predecessor and its successor can service requests for it. Similar to DDS, Petal allows reads to be serviced by either a primary or secondary, but writes must occur at the primary, and data is locked on a write. Hence Petal shares some of the same disadvantages as discussed earlier under DDS. Another key difference between our work and Petal is that although both systems address storage, we address different levels of the storage hierarchy. Petal attempts to present the image of virtual disk to its clients, and hence must maintain and keep consistent metadata for disk block information; in our system, we deal with in memory objects, and need not maintain and keep consistent such metadata.

There are some interesting settings of W, WQ, and R that correspond to work done in previous research. Setting all the variables to 1 is the equivalent of single-copy memory. Setting W and WQ to 2 is roughly equivalent to the in-memory replication scheme adopted by BEA, and setting W to the total number of bricks and R to 1 is the equivalent of "write all, read any."

Future Work:

We hope to explore the effect of faulty or overloaded bricks on overall performance. We expect that the system will degrade gracefully in the presence of n faulty bricks when $W > n + WQ$.

An interesting point to investigate is "shooting bricks," either for garbage collection, or to restart a brick that is performing poorly because it has been running too long. With respect to GC, since session state for a client is rejuvenated on each request, it may be possible simply "shoot" bricks that are reaching capacity, and restart them proactively to avoid garbage collection. A large Internet portal employs a similar strategy, by proactive rebooting its web servers to avoid out-of-memory errors caused by memory leaks. We do not expect rebooting to have a significant impact on performance of the system, given a sufficient number of bricks and an appropriate setting for WQ.

We hope to investigate the effects of the ratio of W to WQ and how performance is affected. We expect that as W grows larger than WQ that the system will perform without bottlenecks, until system capacity is reached.

Conclusion:

This paper argues for a new middle-tier storage layer that handles session state. We believe the storage system avoids the pitfalls of previous solutions, in particular, poor failure/recovery behavior and performance coupling.

References:

[1] Sun Microsystems. Java2 EnterpriseEdition. http://java.sun.com/j2ee/.

[2] U. Singh. Personal communication. E.piphany, 2002.

[3] C.G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles,* pages 2002-210, Litchfield Park, AZ, 1989.

[4] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for iIternet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation,* San Diego, CA, Oct. 2000.

[5] D. Jacobs. Distributed Computing with BEA WebLogic server. In *Proceedings of the Conference on Innovative Data Systems Research,* Asilomar, CA, Jan. 2003.

[6] D. Jacobs. Personal communication, BEA Systems, December 2002.

[7] J. Gray. The Transaction Concept, Virtues and Limitations. In *Proceedings of VLDB,* Cannes, France, Sept 1981.

[9] Network Appliance. http://www.networkappliance.com.

[10] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. SIGPLAN Notices, 32(5):97—108. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* May 1997.

[11] Robert H. Thomas: A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. TODS 4(2): 180-209(1979)

[12] David K. Gifford: Weighted Voting for Replicated Data. Proceedings 7th Symposium on Operating Systems Principles: 150-162, 1979.

[13] G. Candea and A. Fox, Crash-Only Software, Submitted to HotOS 2003.

[14] M. Seltzer and M. Olson. Challenges in embedded database system administration. In Proceeding of the Embedded System Workshop, 1999. Cambridge, MA

[15] Concurrency Control and Recovery in Database Systems, by P.A. Bernstein, V. Hadzilacos and N. Goodman.

[16] Petal, Distributed Virtual Disks, Edward K. Lee and Chandramohan A. Thekkath. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA, 1996), pp. 84--92.

Towards a Semantic-Aware File Store

Zhichen Xu, Magnus Karlsson, Chunqiang Tang* and Christos Karamanolis
HP Laboratories, 1501 Page Mill Rd., MLS 1177, Palo Alto, CA 94304
{zhichen,karlsson,chunqian,christos}@hpl.hp.com

Abstract—Traditional hierarchical namespaces are not sufficient for representing and managing the rich semantics of today's storage systems. In this paper, we discuss the principles of semantic-aware file stores. We identify the requirements of applications and end-users and propose to use a generic data model to capture and represent file semantics. A distinct challenge that we face is to handle dynamic evolution of the data schemas. Further, we outline a framework of basic relations and tools for generating and using semantic metadata. The proposed data model and framework are aimed to be more generic and flexible than what is offered by existing semantic file systems. We envision a range of applications and tools that will exploit semantic information, ranging from personal storage systems with features for advanced searching and roaming access, to enterprise systems supporting distributed data location or archiving.

1 Motivation

Over the last several years, we have witnessed an unprecedented growth of the volume of stored digital data. In 1999, a study estimated the amount of original digital data generated annually to be in excess of 1,700 petabyte [15]. It is estimated that this number has been nearly doubling annually since then [22]. This explosive growth is reflected on the ever increasing complexity and cost for storage management. One instance of this problem occurs in file stores. The traditional hierarchical file system is no longer adequate for systems that need to store billions of files and capture different types of semantic information that is required to efficiently access, share, and manage those files.

Consider, for example, the case of a digital movie production studio. Digital movies consist of hundreds of scenes. Each scene is composed of thousands of different data objects, including character models, backgrounds, and lighting models. These objects are typically implemented as files that are shared by tens of artists. There is a range of semantic information that needs to be captured and used in this environment. When a new version of the hair of a character is created, it has to be annotated with the changes done. Further, it is compatible with only certain versions of the head. Such information about *versions* and *dependencies* among files is important when rendering a scene; it is required to combine objects that are compatible with each other and make sense in some context. When composing a scene, an artist uses material that other people have edited and stored in the system. *Content-based searching* (e.g., search for "green lush grass") as opposed to searching by file name can greatly simplify collaboration and improve productivity. The *view* of what data are stored in the system may potentially be different depending on application and user. For example, an artist wants to see only objects that are compatible with the version of the character she is working on; a backup system only sees files that are marked as "persistent" by the artists. Further, tracking context information, such as the files accessed before, and other statistical information may enable intelligent resource provisioning, data caching and prefetching, and improve search efficiency and accuracy.

Examples of common types of semantic information that needs to be captured include: (i) file versioning, (ii) application-based dependencies, (iii) attribute-based semantics, (iv) content-based semantics, and (v) context-based information.

Considered individually, some of these types of semantic information are captured and used by existing applications and tools, such as versioning control systems or software configuration tools. However, different types of semantic information often depend on each other and are related to other functions of a storage system. For example, application-based dependencies are defined on versions of files. Also, dependencies need to be considered during archiving, to save a consistent snapshot of the application state. We argue that it is easier and more efficient to manage all the above types of semantic information in a single, general-purpose system, that many applications can use.

Along these lines, we propose a *semantic-aware* file store, named *pStore*, that extends file systems—a storage abstraction assumed by many applications—to support semantic metadata. The paper makes the following contributions.

- Proposes using a generic data model to represent semantic information in file systems. The data model has two main features. First, it is extensible to cover semantic information other than the types described above.

*Chunqiang Tang is with Department of Computer Science, University of Rochester, Rochester, NY.

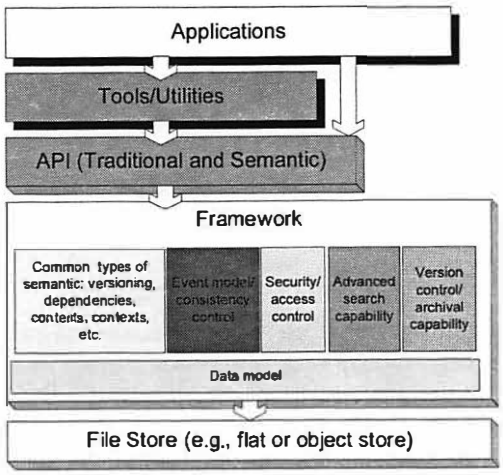


Figure 1: Architecture of pStore.

Second, handles schema evolution, which is essential for many data management applications where semantic information is discovered incrementally.

- Introduces a framework with built-in support for representing and providing access to a set of basic types of semantic information in file systems.
- Outlines a range of applications and tools that can exploit rich semantic information.
- Concludes with a list of research challenges that need to be addressed to realize the vision.

2 Architecture of pStore

The architecture of pStore is illustrated in Figure 1. pStore makes no particular assumption of the underlying file repository, except that it provides a flat space of unique object IDs. The core of pStore is a generic data model that is used to represent semantic information. On top of the data model, a set of basic functionality modules are provided to programmers that wish to develop tools of applications that use or change the semantic data. We describe the basic components of pStore in the following sections.

2.1 Semantic data model

pStore proposes using a generic data model to capture different types of semantic information in file stores. The data model should meet the following requirements.

- Allow to specify well-defined schemata (schema definition language).
- Support dynamic schema evolution to capture new or evolving types of semantic information.
- Be simple to use, lightweight, make no assumptions about the semantics of the metadata.
- Be platform independent and provide interoperability between applications that manage and exchange metadata.
- Facilitate integration with resources outside the file store and support exporting metadata to the web.
- Leverage existing standards and corresponding tools, such as query languages.

Database systems do not fulfill the above requirements, because of two main reasons. First, DBs typically require a predefined schema and impose strict integrity constraints. They cannot effectively deal with incremental and dynamic schema evolution, which is common in managing unstructured data. Second, not all applications require the heavyweight ACID properties and all the features of a fully-fleshed DB. For example, Unix file systems do not guarantee the ACID properties in the face of system failures.

Based on these requirements, we propose using a data model that is based in the *Resource Description Framework* (RDF) [23]. RDF has been proposed to encode, exchange and reuse metadata on the Web (a fundamental tool for realizing the Semantic Web vision [21]). RDF has two main advantages. First, it provides the means to capture schemata for metadata that are both human-readable and machine-processable (RDF notations are typically defined in XML). Second, it is designed to allow reuse and extensions of existing schemata for an ever evolving set of semantic metadata.

RDF is a model that describes resources. Relations, in RDF, are expressed as tuples of the form:

subject `property` *object*

In our case, the subject is a file in the file store. The properties (one or more) that are associated with the subject capture some type of semantic property of the corresponding file. The object of the relation corresponds to the value of the property for the subject, which may be another file or some metadata structure (a literal or composite). Thus, files and metadata structures are both considered resources. In fact, relations themselves can be used as resources for constructing more complex metadata relations.

RDF provides no vocabulary that assumes or refers to application-specific semantic information, e.g., certain properties for media files or relations of files that are accessed by the same user. Instead, such classes of resources and properties are defined in the form of an RDF schema. The same RDF notation is used to specify RDF schemata [24]. This is achieved by providing a set of predefined resources, namely *Classes* and *Properties*. For example, in our case, a Class may refer to files with a certain type of content or files that are used by a certain application. For the model, the specific files are resources that are instances of a certain Class. A Property is defined in the schema to have a *domain* and a *range*. Each of them can be defined to refer to resources of one or more

classes. Classes and Properties can be defined in a hierarchical fashion resulting in schemata that capture complex semantic information.

The principles of RDF resemble those of graph-based data models that have been proposed to handle structural irregularity and incompleteness of schemata and rapid schema evolution [1]. In such systems, the schema is non-mandatory, i.e., it provides some information about the current type of the data, but it does not constrain the format of the data. We have chosen RDF, as it is simple and standardized.

A remaining issue is how to implement a repository of RDF relations in a system. We intend to use some lightweight, RISC-style database systems, like the one proposed by Chaudhui and Weikum [4].

2.2 Basic relations

In the following, we describe a number of relations that cover the set of common types of semantic information listed in Section 1. An RDF schema is defined for each of these relations, but it is not provided here, due to space restrictions. Neither do we use RDF notation to describe relations. Instead, we use an informal triplet notation, as above, using curly brackets to represent composite properties (constructed by means of blank properties or containers in RDF).

File versioning. Each file in pStore corresponds to one *file object* and multiple *file version objects*[1]. Each update to the file automatically creates a new file version. The notion of a "file" will be represented by a data object that captures some of the basic attributes of the file (owner, file name, etc). For example, it could be the root node in a hierarchical content-addressable storage system [17]. As soon as the file has some content, each version of the file is represented by another object.

There are two types of relations between a file and its versions. Relation *o1* `has_version`{*o2, v1*} states that object with id *o2* is version *v1* of *o1*. Similarly, *o1* `latest_version`{*o2*} states that object *o2* is the latest version of *o1*. Property `has_version` may have additional attributes, such as `creation_time`, and `comment`.

Hierarchical name space. The traditional hierarchical name space is defined using the `is_parent_of` and `in_directory` properties. E.g., *"movie1* `is_parent_of` *sequence2"* represents the file path "movie1/Sequence2". File system access control is represented by the `access_control` property. The range of this property is a Class that defines, e.g., an ACL structure.

Dependencies. In addition to the hierarchical relations, a user can define other types of dependencies among objects. In fact, `is_parent_of` is just one instance of Property schema `Depend_on`. Instances of this Property may be application specific. For example, the relation *Shrek* `char_dep` *Ogre*, where `char_dep` is an instance of `Depend_on`, means that file *Shrek* has a dependency on file *Ogre*. Another example of dependency is the relationship between the master copy of the data and its replicas.

Associative semantics. Another common relationship is that of a metadata object describing an ordinary file. For instance, *Fiona* `comments` *text* indicates that object *text* describes the *Fiona* character. Such metadata will, in many cases, be automatically extracted and used for searching, as explained in the next section.

Context information. The data model can also be used to track context information from the file system and user behavior. Examples of related properties include `no_reads`, `no_writes`, `accessed_before`, `accessed_by`, and `accessed_from`. For example, we can use *hair* `accessed_before` {*time=5s, nose*} to record the fact that file *hair* is accessed 5 seconds before accessing file *nose*. This information can be used, to gather statistics that pStore (or applications) can use to improve the performance of the system. Examples include prefetching and caching in distributed environments, data placement, as well as advanced searching.

An important challenge that needs to be addressed is automatically extracting various types of semantic information from data. E.g., people use vector space models to extract features from text documents and images [2, 5]. Similarly, they derive frequency, amplitude, and tempo feature vectors from music data [6]. More recently, Soules and Ganger [19] proposed methods for capturing file attributes and inter-file relations, by analyzing user access patterns.

2.3 Dynamic evolution of schema

We expect pStore to provide a set of default schemata, like the ones above (and possibly more). However, we expect users to modify these schemata. For example, in many data management applications, relationships among data objects are identified after the objects are created and may change during the lifetime of the objects, as their usage changes. The usage of data and metadata is often unpredictable and may depend on the actual user or workload. Incremental elaboration of data object classes and their properties is often inevitable. We also expect users to define their own schemata and share them in ad-hoc manners to cover application or site-specific requirements among communities of users.

RDF supports dynamic evolution of schema in multiple ways. First, it supports refinement of schema through class inheritance and property polymorphism. Second, the *namespace* feature of RDF allows for schemata to evolve differently in different contexts, such as application versions or user communities. Last, but not least, the fact that RDF provides a machine-readable notation, facilitates the design of programmable interfaces and tools that allow for automatic extraction, manipulation and exchange of relations and schemata.

[1] These are data objects, not necessarily related with the object of an RDF relation.

2.4 Framework

The pStore framework offers built-in support for representing and accessing semantic metadata in file stores.

Event model/consistency control. Inter-file dependencies is an important type of semantic information captured by pStore. Often, such dependencies imply some consistency requirement users assume between the related files. Such requirements vary for different instances of a relation, or even across time.

We capture such consistency requirements by augmenting dependency relations with an associated relation of type `Event`. An event consists of an ordered list of ⟨*precondition: action*⟩ tuples (implemented as a `rdf:seq` container in RDF). When a data object is accessed (e.g., open, write), the system checks each of these preconditions and executes the corresponding actions if the precondition holds. Suppose that object *Shrek* depends on object *Ogre*. One of the events associated with that relation may look like ⟨*modified: rebuild(Shrek)*⟩, specifying that *Shrek* needs to be regenerated if *Ogre* is modified.

Customized name space views. In addition to the conventional hierarchical name space, the data model provides the basis on which customized per-user or per-application name spaces can be constructed. We sketch several ways that this can be done.

One way to construct customized name spaces is by constraining the corresponding relations. A special case is when the customized name space is a sub-graph of the original file system hierarchy. For instance, *Shrek* `is_parent_of` {*user=Mary, script*} states that object *Shrek* is a parent directory of object *script* only for user Mary. Another possibility is to exploit Property inheritance in the schema. For example, Property `land_mammal{feet}` can be regarded as a *super class* of Property `elephant{feet, trunk}`.

In principle, a virtual directory can be created to include links to an arbitrary set of files, e.g., searching results [8].

Security and access control. In an enterprise environment such as a digital movie studio, data is its biggest asset. Thus, data dependability is of paramount importance. They use mechanisms such as encryption and access control to protect the data and mechanisms such as erasure coding and replication for high reliability and availability. We envision that such data dependability mechanisms can be represented using our data model. RDF Property inheritance can be used to fine tune the relations for certain types of data.

Advanced searching capabilities. One of the open research questions in storage systems today is how to perform advanced and efficient searching of content in large corpuses of data. Our model and framework provide a uniform platform for integrating *content*, *attribute*, and *context-based* searching. For example, it can be used in combination with information retrieval algorithms [2] that depend on semantic information from the data. Similarly, our model can capture context information (such as access patterns) and inter-file relationships that can be used for advanced context-based searching [19]. We would also like to provide searching with variable recall and precision to be able to trade-off this against speed. Especially for queries where the recall and precision are not 100%, the ranking of the search results becomes important. This is an area where context information has been successfully used, for example in Google.

Archival support. An on-line archival storage system is one of the main applications we envision for pStore. Compression and versioning are essential given the volume and complexity of the data [17]. The semantic information that our model can capture about the data can be used to reduce storage consumption [11] and facilitate efficient data organization for fast data storage and retrieval.

3 Application Scenarios

In the following paragraphs, we describe some examples of applications of pStore other than a digital movie studio to demonstrate the generality of our proposal.

Online data sharing. In general, it is desirable that each object can have an arbitrary metadata structure suitable for describing its contents as well as its relationships with other objects. Objects can relate to each other in many different ways: an object may overlap with or include other objects; multiple objects may share descriptive data. In practice, meaningful objects are often identified and associated with their descriptive data incrementally and dynamically, after the data is stored in the system.

To provide adequate control, users can be given different access privileges. To facilitate collaboration, in addition to a shared global view of all the data, there may also be customized per-user and per-application views. Advanced searching capabilities are needed to allow people to effectively navigate among the various digital components.

A semantic, deep archival system. It is now practically affordable to archive each individual version of a file. Such archival storage system are becoming essential for many critical applications. We list some desirable features.

First, a user would like the file store to have a "travel-in-time" capability-—every change to an object or to the name space is recorded, and a user can travel arbitrary back in time to retrieve any version of a file that ever existed [11]. An important challenge is to maintain the various dependencies among different versions of objects and handle time as yet another type of semantic information.

Second, to reduce storage space consumption, objects should be stored efficiently. Various data clustering and compression techniques are being explored. One way to do this is to exploit the available semantic information. E.g., when generating a new version of a file, the semantic information is used to identify an existing (base) file with similar contents. Only the differences between the new and the base file are stored.

Last, in restoring a backed-up version, the biggest headache is to find the right document and the right version. With pStore's rich metadata model, the semantic information of files can be associated with files. In the restoring operation, the user describes a desired feature that is known to exist in the recovered version. For example, the system may use content extracts to locate the right version, without requiring the user remembering the exact name or creation date of the restored file.

Digital content distribution. In addition to search capabilities, a large-scale distributed file system can utilize the relationships among files to guide data placement, and perform caching and prefetching. CDN more efficient. Another related application is to support data hoarding for mobile users. Before disconnected from the network, all frequently used data for the user are identified through examining the metadata, and are automatically moved to a portable device. Systems such as SEER [10] use simple semantic hints such as user activity and directory membership for hoarding related files. Their effectiveness is limited by operations such as running the UNIX `find` utility across an entire file system.

Personal storage for desktop users. Many of the features described above can benefit ordinary desktop users as well. As desktop users, we would like to keep every version of important files that we ever created or downloaded, add arbitrary annotations to the files, relate them to the their sources, and create cross links among them. Automated file hoarding can relieve much of the pain to manually identify and move files among computers and mobile devices. Many of us have painful experiences of not finding files. The advanced searching capability would make search much easier.

4 Related Work

Contemporary file systems use file type information to associate files with the appropriate applications to access them. Further, several systems have experimented with the idea of attribute-based file naming [7, 8, 13, 16, 18]. The file system supports searching on the basis of attributes; the results are reflected on virtual directories that contain pointers to the actual locations of files.

SFS [7] uses a hierarchical directory structure to organize refinements to previous query results. HAC [8] attempts to combine the benefits of hierarchical and content-based access to files at the same time. A virtual directory (resulting from a query) is an actual directory that allows ordinary file system operations. To maintain the consistency between links in a virtual directory and the files they point to, HAC re-executes queries periodically to update the links in virtual directories.

Several systems allow for more flexible ways to combine the hierarchical name space with attribute-based file naming. A file system by Transarc [3] allows each file to have an associated wrapper, called a synopsis, that contains tag/value attributes and defines methods to manipulate those attributes. Synopses are organized in inheritance hierarchies. Similarly, in a system described in [18], each query is given a label. Users can impose "ancestor-descendant" relationship on labels, and consequently can name files by specifying either the path name that contains labels, or a list of queries the files satisfy, or both. In the Prospero system [13], users can program "filters" that create personalized views of file systems.

In Presto [16], documents can be organized according to properties (attributes) that are associated with the documents, without the limitations of hierarchies. Properties can be specific to an individual document consumer. Unlike HAC, Presto does not intend to handle backward compatibility to the traditional file system abstraction.

All these systems focus mainly on simple attributes; queries are limited to ad-hoc attribute match. pStore provides a generic data model and implementation that capture a more extensive set of semantics. We anticipate that these attributed-based file systems can be easily implemented using pStore and pStore's generality can be explored to provide new functionalities that do not exist in these systems.

Several projects study metadata management in a file system setting. Roma [20] provides an available, centralized repository of metadata to "synchronize" a single user's files across a diversity of digital storage devices. Roma metadata include fully-extensible attributes that could be used for organizing and locating files. However, its current prototype does not utilize attributes for searching.

The Inversion file system [14] runs on top of the POSTGRES database. It allows fine-grained time travel—a user may ask to see the state of the file system at any time in the past. Accesses to the file system are transactional. It is possible to issue ad-hoc queries on the file system metadata, or even to file data. IBM's DataLink [9] project uses a relational database to capture a wide set of semantic information in file systems. The database contains references to objects in the file system. However, not all applications require the heavyweight ACID properties and features of a fully-fleshed database system. Moreover, database systems cannot effectively handle the incremental evolution of schema, common when managing unstructured data.

It is interesting to note that, as early as 1986, Mogul [12] has proposed a model of files that includes the concept of file properties. Mogul also agrees that database systems are too heavyweight, and relationships between files are important.

Our work complements the semantic Web [21] by concentrating on the system aspects and metadata management in a storage setting. Further, pStore provides additional functionality, e.g., tunable consistency based on an event-framework. It is a framework that provides predefined but customizable components. One example is the predefined

types of metadata (e.g., content- and context-based semantics) each possibly with predetermined consistency models.

5 Conclusion and Open Issues

The paper motivates the need to incorporate semantic metadata in file stores. We identify the basic types of semantic information required by applications and end-users and propose a generic data model to capture and represent file semantics. The model provides the basis for a framework of tools and APIs for generating and using semantic metadata. There is a large number of research problems that need to be addressed to realize a semantic-aware file store. We enumerate some of them below.

- The basic semantic relations sketched in section 2.2 are yet to be evaluated and finalized through the use of real applications.
- Investigate the design of semantic-aware deep-archival systems. In particular, what kind of semantic information can be used for improved data clustering and compression techniques. Also, how to maintain rich semantics for multiple versions of files; inheritance of semantic relations and their representation and use.
- Use semantic metadata for intelligent data placement in distributed storage systems. The goal is to satisfy the QoS requirements of end-users or applications with low infrastructure cost.
- Design and implement a basic set of tools and APIs for using the semantic information captured in such systems. These tools should be extensible and customizable. What these tools will be and how they will interact with each other is an open issue.
- Devise a simple declarative query language that can be used to specify constraints on both structured and unstructured data components.
- Investigate how the proposed data model and framework can be implemented in a distributed file system efficiently. One hard question is how to store RDF relations using a lightweight DB.

We are currently implementing a prototype of pStore to demonstrate its benefits in an online archival storage system.

References

[1] S. Abiteboul. Querying semi-structured data. In *6th International Conference in Database Theory - ICDT '97*, pages 1–18, Delphi, Greece, January 1997.

[2] M. Berry, Z. Drmac, and E. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[3] M. Bowman. Managing Diversity in Wide-Area File Systems. In *Second IEEE Metadata Conference*, September 1997.

[4] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *The VLDB Journal*, pages 1–10, 2000.

[5] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[6] J. Foote. An overview of audio information retrieval. *Multimedia Systems*, 7(1):2–10, 1999.

[7] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[8] B. Gopal and U. Manber. Intergrating content-based access machanisms with hierarchical file systems. In *the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, USA, 1999.

[9] H.-I. Hsiao and I. Narang. DLFM: A Transactional Resource Manager. In *SIGMOD Conference 2000*, 2000.

[10] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Symposium on Operating Systems Principles*, pages 264–275, 1997.

[11] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. In *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, May 2003.

[12] J. C. Mogul. *Representing Information About Files.* PhD thesis, Stanford University, March 1986.

[13] B. C. Neuman. The prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, 1992.

[14] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 205–217, San Diego, CA, USA, 25–29 1993.

[15] P. Lyman, H.R. Varian, J. Dunn, A. Strygin, and K. Searingen. How much information, October 2000. http://www.sims.berkeley.edu/research/projects/how-much-info.

[16] A. L. Paul Dourish, W. Keith Edwards and M. Salisbury. Using properties for uniform interaction in the presto document system. In *The 12th Annual ACM Symposium on User Interface Software and Technology*, Asheville, NC, USA, November 7–10 1999.

[17] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey, CA, USA, 2002.

[18] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *12th International Conference on Distributed Computer System*, Yokohama, Japan, June 1992.

[19] G. A. N. Soules and G. R. Ganger. Why can't i find my files? new methods for automating attribute assignment. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, May 18-21 2003.

[20] E. Swierk, E. Kiciman, V. Laviano, and M. Baker. The roma personal metadata service. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, USA, December 2000.

[21] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[22] The Enterprise Storage Group. Reference information: The next wave "the summary of: A snapshot research study by the enterprise storage group", 2002. http://www.enterprisestoragegroup.com.

[23] W3C. Resource description framework (rdf) model and syntax specification, February 22 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[24] W3C. Resource description framework (rdf) schema specification, March 3 1999. http://www.w3.org/TR/1999/PR-rdf-schema-19990303/.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

USENIX Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year. *;login:* is sent in print, and is available on the USENIX Web site to all current USENIX members
- Access to papers from the USENIX conferences and symposia, starting with 1993, on the USENIX Web site
- Discounts on registration fees for the annual, multi-topic technical conference, LISA, the Systems Administration Conference, and the various single-topic symposia and workshops
- Discounts on conference proceedings and CD-ROMs from USENIX
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers
- Savings on books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html*

SAGE

SAGE is a Special Technical Group within USENIX. SAGE is an international membership organization whose purpose is to further advance system administration as a profession. For more information, please visit our Web site: *http://www.sage.org/.*

SAGE Member Benefits

- Discount on the registration fee for the annual LISA conference
- Access to SAGEweb, which offers many Web-based services
- The ability to join SAGE-only electronic mailing lists
- Annual System Administration Salary Survey—an annual survey of system and network administrator salaries and responsibilities—and the results of the surveys
- The SAGE Short Topics series, practical booklets covering system administration issues
- The right to vote for the SAGE Executive Committee and on other SAGE matters
- A Code of Ethics for System Administrators
- Discounts on *Sys Admin* and other publications

(Please note that *;login:* is available only as a benefit of USENIX membership.)

There are several classes of membership. You are welcome to become a member of either USENIX or SAGE, or both. Please visit our Classes of Membership Web page for further information: *http://www.usenix.org/membership/classes.html.*

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Ajava Systems, Inc. ❖ Aptitune Corporation ❖ Atos Origin B.V. ❖

❖ Computer Measurement Group ❖ Interhack Corporation ❖ MacConnection ❖

❖ The Measurement Factory ❖ Microsoft Research ❖ Motorola Australia Software Centre ❖

Sun Microsystems, Inc. ❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

❖ Microsoft Research ❖ Ripe NCC ❖

For more information about membership, conferences, or publications, please visit: *http://www.usenix.org/.*

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-17-X